BOOK 16: REACT FUNDAMENTALS
Chapter 1: What is React

WHY REACT EXISTS

Building user interfaces is hard.

Traditional approach:
- Write HTML
- Manipulate DOM with JavaScript
- Track what changed
- Update the right elements
- Keep everything in sync

This becomes messy as applications grow.

React solves this by:
- Describing what UI should look like
- Automatically updating when data changes
- Breaking UI into reusable pieces
- Handling DOM manipulation for you

WHAT IS REACT

React is a JavaScript library for building user interfaces.

Created by Facebook in 2013.

Core ideas:
- Component-based architecture
- Declarative programming
- Virtual DOM for performance
- One-way data flow

React is not a framework. It focuses only on UI.

You choose other libraries for routing, state management, etc.

COMPONENTS

Components are building blocks of React applications.

Each component is a reusable piece of UI.

```
function Welcome() {
 return <h1>Hello, World!</h1>
}
```

This is a component. It returns what should appear on screen.

Components can be:
- Small (a button)
- Medium (a form)
- Large (a page)

You compose small components into larger ones:

```
function App() {
 return (
  <div>
  <Header />
  <Main />
  <Footer />
  </div>
 )
}
```

App uses Header, Main, and Footer components.

DECLARATIVE VS IMPERATIVE

Imperative (traditional):

Tell the computer HOW to do something.

```
let button = document.createElement("button")
button.textContent = "Click me"
button.addEventListener("click", function() {
 let count = parseInt(button.dataset.count) || 0
 count++
 button.textContent = "Clicked " + count + " times"
 button.dataset.count = count
})
document.body.appendChild(button)
```

Declarative (React):

Tell the computer WHAT you want.

```
function Counter() {
 let [count, setCount] = useState(0)

 return (
 <button onClick={() => setCount(count + 1)}>
 Clicked {count} times
 </button>
 )
}
```

React figures out how to update the DOM.
You describe the end result.

VIRTUAL DOM

React uses a Virtual DOM for performance.

How it works:

1. You describe UI with components
2. React creates virtual DOM (JavaScript objects)
3. When state changes, React creates new virtual DOM
4. React compares old and new virtual DOM
5. React updates only what changed in real DOM

This is called reconciliation.

Benefits:
- Batched updates
- Minimal DOM manipulation
- Predictable updates
- Better performance

JSX

JSX is JavaScript XML. It looks like HTML in JavaScript.

let element = <h1>Hello, World!</h1>

JSX is not HTML. It gets transformed to JavaScript:

let element = React.createElement("h1", null, "Hello, World!")

JSX makes React code readable.
You write what looks like HTML.
Babel transforms it to function calls.

ONE-WAY DATA FLOW

Data flows down in React.

Parent passes data to children.
Children cannot modify parent data directly.
Children notify parent through callbacks.

```
function Parent() {
 let [name, setName] = useState("Alice")

 return <Child name={name} onNameChange={setName} />
}

function Child({ name, onNameChange }) {
 return (
 <input
 value={name}
 onChange={(e) => onNameChange(e.target.value)}
 />
 )
}
```

Parent owns the data.
Child displays and reports changes.
Parent decides what to do.

This makes applications predictable.

REACT ECOSYSTEM

React itself is just the UI library.

Common additions:

Routing:
React Router for navigation between pages.

State Management:
Redux, Zustand, or Context for global state.

Styling:
CSS Modules, Styled Components, Tailwind CSS.

Data Fetching:
React Query, SWR, or fetch/axios.

Forms:
React Hook Form, Formik.

Build Tools:
Vite, Create React App, Next.js.

WHERE REACT RUNS

Browser (client-side):
Most common. React runs in browser.
JavaScript builds and updates UI.

Server (server-side rendering):
React runs on server.
Server sends HTML to browser.
Faster initial load.

Static (static site generation):
React builds static HTML at build time.
Great for blogs, documentation.

Mobile (React Native):
Same concepts for iOS and Android apps.

WHEN TO USE REACT

Good for:
- Single-page applications

- Complex user interfaces
- Applications with lots of state
- Reusable component libraries
- Teams needing consistent patterns

Maybe not needed for:
- Simple static websites
- Minimal interactivity
- SEO-critical content (unless using SSR)
- Very small projects

THINKING IN REACT

Building with React is different.

1. Break UI into components
 What are the pieces?
 What can be reused?

2. Build a static version first
 Components that render data.
 No interactivity yet.

3. Identify the state
 What data changes?
 What triggers changes?

4. Determine where state lives
 Which component owns the state?
 Usually the common ancestor.

5. Add inverse data flow
 Child components that notify parents.
 Callbacks passed as props.

PRACTICE

Understand why React exists.

Learn the core concepts.

Think about UIs as components.

Prepare for hands-on coding.

WHAT YOU LEARNED

You learned why React was created.
You learned about components.
You learned declarative vs imperative.
You learned about Virtual DOM.
You learned about JSX.
You learned one-way data flow.

The next chapter teaches you how to set up React.
BOOK 16: REACT FUNDAMENTALS
Chapter 2: Setting Up React

WAYS TO START REACT

Several options for starting a React project:

1. Vite (recommended)
 Fast, modern build tool.
 Quick development server.

2. Create React App

Official React starter.
Zero configuration.
Being phased out.

3. Next.js
Full framework with routing.
Server-side rendering.

4. CDN links
Quick experiments.
No build step.

We will use Vite.

SETTING UP WITH VITE

Prerequisites:
- Node.js installed
- npm or yarn

Create new project:

npm create vite@latest my-react-app -- --template react

Breaking down:

npm create vite@latest - Run Vite project creator
my-react-app - Your project folder name
--template react - Use React template

Navigate to project:

cd my-react-app

Install dependencies:

npm install

Start development server:

npm run dev

Open browser to http://localhost:5173

PROJECT STRUCTURE

my-react-app/
 node_modules/
 public/
 src/
 App.css
 App.jsx
 index.css
 main.jsx
 index.html
 package.json
 vite.config.js

node_modules - Installed packages
public - Static files (images, fonts)
src - Your React code
index.html - HTML entry point
package.json - Project configuration
vite.config.js - Vite configuration

KEY FILES

index.html:

<!DOCTYPE html>
<html lang="en">

```
<head>
 <meta charset="UTF-8">
 <meta name="viewport" content="width=device-width, initial-
scale=1.0">
 <title>My React App</title>
</head>
<body>
 <div id="root"></div>
 <script type="module" src="/src/main.jsx"></script>
</body>
</html>
```

div id="root" is where React renders.
Script loads the React application.

src/main.jsx:

```
import React from "react"
import ReactDOM from "react-dom/client"
import App from "./App"
import "./index.css"

ReactDOM.createRoot(document.getElementById("root")).render(
 <React.StrictMode>
 <App />
 </React.StrictMode>
)
```

Breaking down:

import React - React library
import ReactDOM - React DOM renderer
import App - Your main component
ReactDOM.createRoot - Create React root
getElementById("root") - Find the container
render - Display the App component
React.StrictMode - Development checks

src/App.jsx:

```
function App() {
 return (
 <div>
 <h1>Hello React!</h1>
 </div>
 )
}

export default App
```

This is your main component.
Everything starts here.

JSX FILE EXTENSION

.jsx vs .js:

Both work for React components.
.jsx indicates file contains JSX.
Convention helps editors and tools.

Use .jsx for components.
Use .js for pure JavaScript files.

YOUR FIRST COMPONENT

Edit src/App.jsx:

```
function App() {
 return (
 <div>
 <h1>My First React App</h1>
```

```
 <p>Welcome to React!</p>
 </div>
 )
}

export default App
```

Save the file.
Browser updates automatically.
This is hot module replacement (HMR).

CREATING MORE COMPONENTS

Create src/components/Header.jsx:

```
function Header() {
 return (
 <header>
 <h1>My Website</h1>
 <nav>
 <a href="/">Home</a>
 <a href="/about">About</a>
 </nav>
 </header>
 )
}

export default Header
```

Use in App.jsx:

```
import Header from "./components/Header"

function App() {
 return (
 <div>
```

```
 <Header />
 <main>
 <p>Welcome to my website!</p>
 </main>
 </div>
 )
}

export default App
```

NPM SCRIPTS

package.json scripts:

```
{
 "scripts": {
 "dev": "vite",
 "build": "vite build",
 "preview": "vite preview"
 }
}
```

npm run dev - Start development server
npm run build - Build for production
npm run preview - Preview production build

ADDING DEPENDENCIES

Install packages with npm:

npm install axios

Use in code:

import axios from "axios"

Common React packages:

npm install react-router-dom
npm install @tanstack/react-query
npm install zustand

ENVIRONMENT VARIABLES

Create .env file:

VITE_API_URL=http://api.example.com
VITE_APP_NAME=My App

Use in code:

let apiUrl = import.meta.env.VITE_API_URL
let appName = import.meta.env.VITE_APP_NAME

Variables must start with VITE_ to be exposed.

FOLDER ORGANIZATION

Suggested structure:

src/
 components/
 Button.jsx
 Header.jsx
 Footer.jsx
 pages/
 Home.jsx

About.jsx
hooks/
useAuth.js
useFetch.js
utils/
helpers.js
api.js
App.jsx
main.jsx

components - Reusable UI components
pages - Page-level components
hooks - Custom React hooks
utils - Helper functions

ABSOLUTE IMPORTS

Configure in vite.config.js:

```
import { defineConfig } from "vite"
import react from "@vitejs/plugin-react"
import path from "path"

export default defineConfig({
 plugins: [react()],
 resolve: {
 alias: {
 "@": path.resolve(__dirname, "./src")
 }
 }
})
```

Now import like:

```
import Header from "@/components/Header"
```

Instead of:

import Header from "../../components/Header"

DEVELOPMENT TOOLS

React Developer Tools:
Browser extension for Chrome and Firefox.
Inspect component tree.
View props and state.
Profile performance.

Install from browser extension store.

BUILDING FOR PRODUCTION

Build optimized version:

npm run build

Creates dist/ folder with:
- Minified JavaScript
- Optimized CSS
- Static assets

Preview the build:

npm run preview

Deploy dist/ folder to hosting.

PRACTICE

Create a new React project with Vite.

Explore the project structure.

Create a few components.

Import and use components.

Install React Developer Tools.

WHAT YOU LEARNED

You learned to create React projects.
You learned project structure.
You learned key configuration files.
You learned to create components.
You learned npm scripts.
You learned environment variables.

The next chapter teaches JSX basics.
BOOK 16: REACT FUNDAMENTALS
Chapter 3: JSX Basics

WHAT IS JSX

JSX is JavaScript XML.
It lets you write HTML-like syntax in JavaScript.

```
let element = <h1>Hello, World!</h1>
```

JSX is not a string.
JSX is not HTML.

JSX is a syntax extension.

Babel transforms JSX to:

```
let element = React.createElement("h1", null, "Hello, World!")
```

WHY JSX

JSX makes React code readable.

Without JSX:

```
React.createElement(
 "div",
 { className: "container" },
 React.createElement("h1", null, "Title"),
 React.createElement("p", null, "Content")
)
```

With JSX:

```
<div className="container">
 <h1>Title</h1>
 <p>Content</p>
</div>
```

Both produce the same result.
JSX is easier to read and write.

JSX RULES

1. Return single root element:

Bad:

```
function App() {
 return (
  <h1>Title</h1>
  <p>Content</p>
 )
}
```

Good:

```
function App() {
 return (
  <div>
  <h1>Title</h1>
  <p>Content</p>
  </div>
 )
}
```

Or use Fragment:

```
function App() {
 return (
  <>
  <h1>Title</h1>
  <p>Content</p>
  </>
 )
}
```

Fragment (<></>) groups without extra DOM element.

2. Close all tags:

Self-closing tags must end with />

```
<img src="photo.jpg" />
<input type="text" />
<br />
```

3. camelCase for attributes:

HTML class becomes className.
HTML for becomes htmlFor.
onclick becomes onClick.
tabindex becomes tabIndex.

```
<div className="container">
 <label htmlFor="email">Email</label>
 <input id="email" onClick={handleClick} tabIndex={1} />
</div>
```

EMBEDDING JAVASCRIPT

Use curly braces {} for JavaScript expressions.

Variables:

```
let name = "Alice"

function Greeting() {
 return <h1>Hello, {name}!</h1>
}
```

Expressions:

```
function Math() {
 return <p>2 + 2 = {2 + 2}</p>
}
```

Function calls:

```
function formatName(user) {
```

```
 return user.firstName + " " + user.lastName
}

let user = { firstName: "Alice", lastName: "Smith" }

function Greeting() {
 return <h1>Hello, {formatName(user)}!</h1>
}
```

Object properties:

```
let user = { name: "Alice", age: 25 }

function Profile() {
 return (
 <div>
 <p>Name: {user.name}</p>
 <p>Age: {user.age}</p>
 </div>
 )
}
```

CONDITIONAL RENDERING

Using ternary operator:

```
function Greeting({ isLoggedIn }) {
 return (
 <div>
 {isLoggedIn ? <p>Welcome back!</p> : <p>Please log in</
p>}
 </div>
 )
}
```

Using logical AND:

```
function Notification({ count }) {
 return (
 <div>
 {count > 0 && <p>You have {count} messages</p>}
 </div>
 )
}
```

If count > 0 is true, renders the paragraph.
If false, renders nothing.

Storing in variables:

```
function Greeting({ isLoggedIn }) {
 let message

 if (isLoggedIn) {
 message = <p>Welcome back!</p>
 } else {
 message = <p>Please log in</p>
 }

 return <div>{message}</div>
}
```

LISTS

Map arrays to elements:

```
function TodoList({ todos }) {
 return (
 <ul>
 {todos.map(todo => (
 <li key={todo.id}>{todo.text}</li>
```

```
 ))}
 </ul>
 )
}
```

key prop is required for lists.
Keys help React track which items changed.

STYLES

Inline styles use objects:

```
let styles = {
 color: "blue",
 fontSize: "20px",
 backgroundColor: "#f0f0f0"
}

function StyledText() {
 return <p style={styles}>Styled text</p>
}
```

Or inline:

```
function StyledText() {
 return (
 <p style={{ color: "blue", fontSize: "20px" }}>
 Styled text
 </p>
 )
}
```

Note double braces: outer for JSX, inner for object.

CSS properties use camelCase:
- background-color becomes backgroundColor

- font-size becomes fontSize
- margin-top becomes marginTop

CLASS NAMES

Use className, not class:

```
function Button() {
 return <button className="btn btn-primary">Click</button>
}
```

Dynamic classes:

```
function Button({ isActive }) {
 let className = "btn"

 if (isActive) {
 className += " active"
 }

 return <button className={className}>Click</button>
}
```

Template literal:

```
function Button({ isActive }) {
 return (
 <button className={`btn ${isActive ? "active" : ""}`}>
 Click
 </button>
 )
}
```

ATTRIBUTES

Standard HTML attributes:

```
<input type="text" placeholder="Enter name" disabled={true} />
```

```
<a href="https://example.com" target="_blank" rel="noopener
noreferrer">
 Link
</a>
```

```
<img src="/photo.jpg" alt="Description" width={200}
height={150} />
```

Boolean attributes:

```
<input disabled />
<input disabled={true} />
<input disabled={false} />
```

Just the attribute name means true.
Omitting means false (or use false explicitly).

SPREADING PROPS

Spread operator for attributes:

```
let buttonProps = {
 type: "submit",
 className: "btn",
 disabled: false
}

function Button() {
```

```
 return <button {...buttonProps}>Submit</button>
}
```

Common pattern:

```
function Input(props) {
 return <input {...props} />
}
```

COMMENTS IN JSX

JavaScript comments inside JSX:

```
function App() {
 return (
 <div>
 {/* This is a comment */}
 <h1>Title</h1>
 {/*
 Multi-line
 comment
 */}
 <p>Content</p>
 </div>
 )
}
```

FRAGMENTS

Fragments group without extra DOM:

```
import { Fragment } from "react"
```

```
function List() {
 return (
  <Fragment>
  <li>Item 1</li>
  <li>Item 2</li>
  <li>Item 3</li>
  </Fragment>
 )
}
```

Short syntax:

```
function List() {
 return (
  <>
  <li>Item 1</li>
  <li>Item 2</li>
  <li>Item 3</li>
  </>
 )
}
```

Fragment with key:

```
function List({ items }) {
 return items.map(item => (
  <Fragment key={item.id}>
  <dt>{item.term}</dt>
  <dd>{item.definition}</dd>
  </Fragment>
 ))
}
```

Short syntax cannot have key.

DANGEROUSLY SET HTML

Render raw HTML (use carefully):

```
function Content({ html }) {
 return <div dangerouslySetInnerHTML={{ _html: html }} />
}
```

Warning: This can expose XSS vulnerabilities.
Only use with trusted content.

PRACTICE

Write JSX with multiple elements.

Embed JavaScript expressions.

Use conditional rendering.

Render lists with keys.

Apply inline styles and classes.

WHAT YOU LEARNED

You learned what JSX is.
You learned JSX rules.
You learned to embed JavaScript.
You learned conditional rendering.
You learned lists and keys.
You learned styles and classes.

The next chapter teaches components in depth.
BOOK 16: REACT FUNDAMENTALS
Chapter 4: Components

WHAT ARE COMPONENTS

Components are the building blocks of React.

Each component is:
- A piece of UI
- Reusable
- Self-contained
- Composable

You build complex UIs by combining simple components.

FUNCTION COMPONENTS

Modern React uses function components:

```
function Welcome() {
 return <h1>Welcome to React!</h1>
}
```

Function name is the component name.
Must start with capital letter.
Returns JSX.

Using the component:

```
function App() {
 return (
  <div>
  <Welcome />
```

```
 </div>
 )
}
```

<Welcome /> renders the Welcome component.

COMPONENT FILES

Each component typically gets its own file.

src/components/Button.jsx:

```
function Button() {
 return <button className="btn">Click me</button>
}

export default Button
```

Using it:

```
import Button from "./components/Button"

function App() {
 return (
 <div>
 <Button />
 </div>
 )
}
```

COMPONENT COMPOSITION

Build complex UIs from simple components:

```
function Header() {
 return (
 <header>
 <Logo />
 <Navigation />
 <SearchBar />
 </header>
 )
}

function Logo() {
 return  <img src="/logo.png" alt="Logo" />
}

function Navigation() {
 return (
 <nav>
 <a href="/">Home</a>
 <a href="/about">About</a>
 <a href="/contact">Contact</a>
 </nav>
 )
}

function SearchBar() {
 return (
 <form>
 <input type="text" placeholder="Search..." />
 <button type="submit">Search</button>
 </form>
 )
}
```

Header composes Logo, Navigation, and SearchBar.

COMPONENT HIERARCHY

React apps form a tree of components:

App
□□□ Header
□ □□□ Logo
□ □□□ Navigation
□ □□□ SearchBar
□□□ Main
□ □□□ Sidebar
□ □□□ Content
□ □□□ ArticleList
□ □ □□□ Article (many)
□ □□□ Pagination
□□□ Footer
□□□ Copyright
□□□ SocialLinks

Each component manages its own piece of UI.

REUSABLE COMPONENTS

Components can be used multiple times:

```
function Card({ title, content }) {
 return (
 <div className="card">
 <h2>{title}</h2>
 <p>{content}</p>
 </div>
 )
}

function App() {
 return (
 <div>
 <Card title="First Card" content="First content" />
```

```
 <Card title="Second Card" content="Second content" />
 <Card title="Third Card" content="Third content" />
 </div>
 )
}
```

Same Card component, different data.

CHILDREN PROP

Components can wrap other content:

```
function Container({ children }) {
 return <div className="container">{children}</div>
}

function App() {
 return (
 <Container>
 <h1>Title</h1>
 <p>Content goes here</p>
 </Container>
 )
}
```

children is whatever goes between opening and closing tags.

Multiple children:

```
function Layout({ children }) {
 return (
 <div className="layout">
 <Header />
 <main>{children}</main>
 <Footer />
 </div>
```

```
 )
}

function HomePage() {
 return (
  <Layout>
  <h1>Home</h1>
  <p>Welcome to our site!</p>
  </Layout>
 )
}
```

NAMED EXPORTS

Export multiple components from one file:

```
export function Button() {
 return  <button>Click</button>
}

export function Input() {
 return  <input type="text" />
}

export function Label({ children }) {
 return  <label>{children}</label>
}
```

Import with destructuring:

```
import { Button, Input, Label } from "./components/FormElements"
```

DEFAULT VS NAMED EXPORTS

Default export (one per file):

```
function Button() {
 return  <button>Click</button>
}

export default Button
```

Import:

```
import Button from "./Button"
import MyButton from "./Button"
```

Named export (multiple per file):

```
export function Button() {
 return  <button>Click</button>
}
```

Import (must use exact name):

```
import { Button } from "./Button"
```

COMPONENT NAMING CONVENTIONS

Components: PascalCase

```
function UserProfile() { }
function ShoppingCart() { }
function NavigationMenu() { }
```

Files: Match component name or kebab-case

UserProfile.jsx

shopping-cart.jsx
NavigationMenu.jsx

Props: camelCase

```
<UserCard userName="alice" isActive={true}
onClick={handleClick} />
```

PURE COMPONENTS

Components should be pure functions:

Same input (props) produces same output (JSX).
No side effects during render.

Pure (good):

```
function Greeting({ name }) {
 return <h1>Hello, {name}!</h1>
}
```

Impure (bad):

```
let count = 0

function Counter() {
 count++
 return <p>Rendered {count} times</p>
}
```

Modifying external state during render is bad.

COMPONENT BEST PRACTICES

1. Single responsibility:
Each component does one thing well.

2. Keep components small:
If a component is too big, split it.

3. Descriptive names:
Names should explain what component does.

4. Props for configuration:
Pass data and callbacks through props.

5. Avoid deep nesting:
Extract nested JSX into components.

Before (deeply nested):

```
function App() {
 return (
  <div className="app">
  <div className="header">
  <div className="logo">
  <img src="/logo.png" alt="Logo" />
  </div>
  <div className="nav">
  <ul>
  <li><a href="/">Home</a></li>
  <li><a href="/about">About</a></li>
  </ul>
  </div>
  </div>
  <div className="content">
 ...
  </div>
  </div>
 )
}
```

After (composed):

```
function App() {
 return (
 <div className="app">
 <Header />
 <Content />
 </div>
 )
}

function Header() {
 return (
 <div className="header">
 <Logo />
 <Nav />
 </div>
 )
}
```

ORGANIZING COMPONENTS

By type:

```
src/
 components/
 buttons/
 Button.jsx
 IconButton.jsx
 forms/
 Input.jsx
 Select.jsx
 layout/
 Header.jsx
 Footer.jsx
```

By feature:

src/
features/
auth/
LoginForm.jsx
SignupForm.jsx
products/
ProductList.jsx
ProductCard.jsx
cart/
Cart.jsx
CartItem.jsx

PRACTICE

Create function components.

Compose components together.

Use children prop.

Practice default and named exports.

Organize components in folders.

WHAT YOU LEARNED

You learned function components.
You learned component composition.
You learned component hierarchy.
You learned the children prop.

You learned export patterns.
You learned component best practices.

The next chapter teaches props.
BOOK 16: REACT FUNDAMENTALS
Chapter 5: Props

WHAT ARE PROPS

Props (properties) are how components receive data.

Parent passes data to child through props.
Props are read-only in the child.

```
function Greeting({ name }) {
 return <h1>Hello, {name}!</h1>
}

function App() {
 return <Greeting name="Alice" />
}
```

name="Alice" is a prop.
Greeting receives { name: "Alice" }.

PASSING PROPS

Props look like HTML attributes:

```
<Component propName={value} />
```

String props:

<Greeting name="Alice" />

Number props:

<Counter count={5} />

Boolean props:

<Button disabled={true} />
<Button disabled />

Object props:

<UserCard user={{ name: "Alice", age: 25 }} />

Array props:

<List items={["Apple", "Banana", "Cherry"]} />

Function props:

<Button onClick={() => console.log("clicked")} />

RECEIVING PROPS

Props come as an object:

```
function Greeting(props) {
 return <h1>Hello, {props.name}!</h1>
}
```

Destructuring (cleaner):

```
function Greeting({ name }) {
 return <h1>Hello, {name}!</h1>
}
```

Multiple props:

```
function UserCard({ name, email, age }) {
 return (
 <div className="user-card">
 <h2>{name}</h2>
 <p>Email: {email}</p>
 <p>Age: {age}</p>
 </div>
 )
}

<UserCard name="Alice" email="alice@example.com" age={25}
/>
```

DEFAULT PROPS

Provide defaults for optional props:

```
function Button({ text = "Click me", type = "button" }) {
 return <button type={type}>{text}</button>
}

<Button />
<Button text="Submit" type="submit" />
```

Or with defaultProps:

```
function Button({ text, type }) {
 return <button type={type}>{text}</button>
}

Button.defaultProps = {
 text: "Click me",
 type: "button"
}
```

PROP TYPES

Validate props in development:

npm install prop-types

```
import PropTypes from "prop-types"

function UserCard({ name, email, age }) {
 return (
 <div>
 <h2>{name}</h2>
 <p>{email}</p>
 <p>{age}</p>
 </div>
 )
}

UserCard.propTypes = {
 name: PropTypes.string.isRequired,
 email: PropTypes.string.isRequired,
 age: PropTypes.number
}
```

Common PropTypes:

PropTypes.string

PropTypes.number
PropTypes.bool
PropTypes.array
PropTypes.object
PropTypes.func
PropTypes.node
PropTypes.element
PropTypes.oneOf(["option1", "option2"])
PropTypes.arrayOf(PropTypes.string)
PropTypes.shape({ name: PropTypes.string })

SPREADING PROPS

Pass all props at once:

```
let userProps = {
 name: "Alice",
 email: "alice@example.com",
 age: 25
}

<UserCard {...userProps} />
```

Same as:

```
<UserCard name="Alice" email="alice@example.com" age={25}
/>
```

Forwarding props:

```
function WrappedInput(props) {
 return (
 <div className="input-wrapper">
 <input {...props} />
 </div>
```

```
 )
}

<WrappedInput type="text" placeholder="Enter name" />
```

REST PROPS

Get remaining props:

```
function Button({ className, children, ...rest }) {
 return (
  <button className={`btn ${className}`} {...rest}>
 {children}
  </button>
 )
}

<Button className="primary" onClick={handleClick}
disabled={false}>
 Click me
</Button>
```

className and children are extracted.
Everything else goes to ...rest.

CHILDREN PROP

Special prop for nested content:

```
function Card({ children }) {
 return  <div className="card">{children}</div>
}

<Card>
```

```
 <h2>Title</h2>
 <p>Content</p>
</Card>
```

Children can be anything:

```
<Card>Text content</Card>
<Card><span>Element</span></Card>
<Card>{items.map(i => <Item key={i.id} />)}</Card>
```

Checking children:

```
import { Children } from "react"

function List({ children }) {
 let count = Children.count(children)

 return (
 <div>
 <p>{count} items</p>
 <ul>{children}</ul>
 </div>
 )
}
```

RENDER PROPS

Pass a function as prop that returns JSX:

```
function DataFetcher({ url, render }) {
 let [data, setData] = useState(null)

 useEffect(() => {
 fetch(url)
 .then(res => res.json())
```

```
 .then(setData)
 }, [url])

 return render(data)
}

<DataFetcher
 url="/api/users"
 render={(data) => (
 data ? <UserList users={data} /> : <Loading />
 )}
/>
```

Or using children as function:

```
function DataFetcher({ url, children }) {
 let [data, setData] = useState(null)

 useEffect(() => {
 fetch(url)
 .then(res => res.json())
 .then(setData)
 }, [url])

 return children(data)
}

<DataFetcher url="/api/users">
 {(data) => data ? <UserList users={data} /> : <Loading />}
</DataFetcher>
```

CALLBACK PROPS

Pass functions for communication:

```
function Child({ onButtonClick }) {
```

```
 return <button onClick={onButtonClick}>Click me</button>
}

function Parent() {
 function handleChildClick() {
 console.log("Child button was clicked!")
 }

 return <Child onButtonClick={handleChildClick} />
}
```

With data:

```
function Child({ onSubmit }) {
 let [value, setValue] = useState("")

 return (
 <form onSubmit={(e) => {
 e.preventDefault()
 onSubmit(value)
 }}>
 <input value={value} onChange={(e) =>
setValue(e.target.value)} />
 <button type="submit">Submit</button>
 </form>
 )
}

function Parent() {
 function handleSubmit(value) {
 console.log("Received:", value)
 }

 return <Child onSubmit={handleSubmit} />
}
```

PROPS ARE READ-ONLY

Never modify props:

```
function Greeting({ name }) {
 name = "Bob"
 return <h1>Hello, {name}!</h1>
}
```

This is wrong. Props should be treated as immutable.

If you need to change data, use state.

LIFTING STATE UP

When siblings need to share data:

```
function Parent() {
 let [value, setValue] = useState("")

 return (
 <div>
 <Input value={value} onChange={setValue} />
 <Display value={value} />
 </div>
 )
}

function Input({ value, onChange }) {
 return (
 <input
 value={value}
 onChange={(e) => onChange(e.target.value)}
 />
 )
}
```

```
function Display({ value }) {
 return <p>You typed: {value}</p>
}
```

State lives in Parent.
Both children access it through props.

PRACTICE

Pass different types of props.

Use destructuring for props.

Set default values.

Use spread operator.

Pass callback functions.

WHAT YOU LEARNED

You learned to pass props.
You learned to receive and destructure props.
You learned default props.
You learned prop types validation.
You learned spread and rest patterns.
You learned callback props.

The next chapter teaches state with useState.
BOOK 16: REACT FUNDAMENTALS
Chapter 6: State with useState

WHAT IS STATE

State is data that changes over time.

Props come from parent and are read-only.
State belongs to the component and can change.

When state changes, React re-renders the component.

WHY STATE MATTERS

Without state, UI would be static.

State enables:
- User interactions
- Form inputs
- Toggling visibility
- Loading indicators
- Counters
- Any dynamic behavior

THE USESTATE HOOK

useState is a React Hook for adding state.

```
import { useState } from "react"

function Counter() {
 let [count, setCount] = useState(0)

 return (
 <div>
```

```
 <p>Count: {count}</p>
 <button onClick={() => setCount(count + 1)}>
 Increment
 </button>
 </div>
 )
}
```

Breaking down:

useState(0) - Initialize state with value 0
[count, setCount] - Destructure the returned array
count - Current state value
setCount - Function to update state

USESTATE SYNTAX

```
let [state, setState] = useState(initialValue)
```

state - Current value
setState - Update function
initialValue - Starting value

The initial value is only used on first render.

UPDATING STATE

Call the setter function:

```
setCount(5)
```

State updates are asynchronous.
React batches updates for performance.

Using previous state:

setCount(count + 1)

Or with callback (safer):

setCount(prevCount => prevCount + 1)

Callback form is better when update depends on previous value.

MULTIPLE STATE VARIABLES

```
function Form() {
 let [name, setName] = useState("")
 let [email, setEmail] = useState("")
 let [age, setAge] = useState(0)

 return (
 <form>
 <input
 value={name}
 onChange={(e) => setName(e.target.value)}
 />
 <input
 value={email}
 onChange={(e) => setEmail(e.target.value)}
 />
 <input
 type="number"
 value={age}
 onChange={(e) => setAge(parseInt(e.target.value))}
 />
 </form>
 )
}
```

Each piece of state is independent.

OBJECT STATE

State can be an object:

```
function Form() {
 let [form, setForm] = useState({
 name: "",
 email: "",
 age: 0
 })

 function handleChange(e) {
 setForm({
 ...form,
 [e.target.name]: e.target.value
 })
 }

 return (
 <form>
 <input name="name" value={form.name}
onChange={handleChange} />
 <input name="email" value={form.email}
onChange={handleChange} />
 <input name="age" value={form.age}
onChange={handleChange} />
 </form>
 )
}
```

Important: Spread the old object, then override the field.
React compares by reference, not by content.

ARRAY STATE

Adding to array:

```
let [items, setItems] = useState([])

function addItem(item) {
 setItems([...items, item])
}
```

Removing from array:

```
function removeItem(id) {
 setItems(items.filter(item => item.id !== id))
}
```

Updating item in array:

```
function updateItem(id, newData) {
 setItems(items.map(item =>
 item.id === id ? { ...item, ...newData } : item
 ))
}
```

Complete example:

```
function TodoList() {
 let [todos, setTodos] = useState([])
 let [text, setText] = useState("")

 function addTodo() {
 if (!text.trim()) return

 setTodos([
 ...todos,
 { id: Date.now(), text, completed: false }
```

```
])
setText("")
}

function toggleTodo(id) {
setTodos(todos.map(todo =>
todo.id === id ? { ...todo, completed: !todo.completed } : todo
))
}

function deleteTodo(id) {
setTodos(todos.filter(todo => todo.id !== id))
}

return (
<div>
<input
value={text}
onChange={(e) => setText(e.target.value)}
placeholder="Add todo"
/>
<button onClick={addTodo}>Add</button>

<ul>
{todos.map(todo => (
<li key={todo.id}>
<input
type="checkbox"
checked={todo.completed}
onChange={() => toggleTodo(todo.id)}
/>
<span style={{
textDecoration: todo.completed ? "line-through" : "none"
}}>
{todo.text}
</span>
<button onClick={() => deleteTodo(todo.id)}>Delete</
button>
</li>
))}
```

```
 </ul>
 </div>
 )
}
```

LAZY INITIALIZATION

For expensive initial values:

```
let [data, setData] = useState(() => {
 return expensiveComputation()
})
```

Function only runs on first render.

Example:

```
let [items, setItems] = useState(() => {
 let saved = localStorage.getItem("items")
 return saved ? JSON.parse(saved) : []
})
```

BOOLEAN STATE

Toggle pattern:

```
function Toggle() {
 let [isOn, setIsOn] = useState(false)

 return (
 <button onClick={() => setIsOn(!isOn)}>
{isOn ? "ON" : "OFF"}
 </button>
```

```
 )
}
```

Or with callback:

```
<button onClick={() => setIsOn(prev => !prev)}>
```

CONTROLLED INPUTS

Input value controlled by state:

```
function TextInput() {
 let [value, setValue] = useState("")

 return (
  <input
 value={value}
 onChange={(e) => setValue(e.target.value)}
 />
 )
}
```

Why controlled:
- Single source of truth
- Validate on every change
- Format as user types
- Clear programmatically

STATE UPDATE BATCHING

React batches state updates:

```
function Counter() {
 let [count, setCount] = useState(0)

 function handleClick() {
 setCount(count + 1)
 setCount(count + 1)
 setCount(count + 1)
 }

 return <button onClick={handleClick}>{count}</button>
}
```

This only increments by 1, not 3.
All three use the same count value.

Fix with callback:

```
function handleClick() {
 setCount(prev => prev + 1)
 setCount(prev => prev + 1)
 setCount(prev => prev + 1)
}
```

Now increments by 3.

STATE VS PROPS

Props:
- Passed from parent
- Read-only
- Parent controls

State:
- Belongs to component
- Can change
- Component controls

Use props for configuration.
Use state for data that changes.

COMMON PATTERNS

Loading state:

```
let [loading, setLoading] = useState(false)
let [data, setData] = useState(null)

async function fetchData() {
 setLoading(true)
 let result = await fetch("/api/data")
 let json = await result.json()
 setData(json)
 setLoading(false)
}
```

Error state:

```
let [error, setError] = useState(null)

async function fetchData() {
 try {
 let result = await fetch("/api/data")
 let json = await result.json()
 setData(json)
 } catch (err) {
 setError(err.message)
 }
}
```

Combined:

```
let [state, setState] = useState({
 loading: false,
 data: null,
 error: null
})
```

PRACTICE

Create a counter component.

Build a form with multiple inputs.

Manage array state (add, remove, update).

Use controlled inputs.

Practice state update patterns.

WHAT YOU LEARNED

You learned what state is.
You learned useState hook.
You learned to update state.
You learned object and array state.
You learned controlled inputs.
You learned state update batching.

The next chapter teaches handling events.
BOOK 16: REACT FUNDAMENTALS
Chapter 7: Handling Events

EVENTS IN REACT

React events are similar to DOM events.

Key differences:
- camelCase names (onClick not onclick)
- Pass function, not string
- Prevent default differently

DOM:

```
<button onclick="handleClick()">Click</button>
```

React:

```
<button onClick={handleClick}>Click</button>
```

BASIC EVENT HANDLING

```
function Button() {
 function handleClick() {
 console.log("Button clicked!")
 }

 return <button onClick={handleClick}>Click me</button>
}
```

Inline handler:

```
function Button() {
 return (
 <button onClick={() => console.log("Clicked!")}>
 Click me
 </button>
 )
```

}

COMMON EVENTS

Click:

```
<button onClick={handleClick}>Click</button>

<div onClick={handleDivClick}>Clickable div</div>
```

Change (inputs):

```
<input onChange={handleChange} />

<select onChange={handleSelectChange}>
 <option>Option 1</option>
</select>
```

Submit (forms):

```
<form onSubmit={handleSubmit}>
 <button type="submit">Submit</button>
</form>
```

Focus and Blur:

```
<input onFocus={handleFocus} onBlur={handleBlur} />
```

Keyboard:

```
<input onKeyDown={handleKeyDown}
onKeyUp={handleKeyUp} />
```

Mouse:

```
<div
 onMouseEnter={handleMouseEnter}
 onMouseLeave={handleMouseLeave}
 onMouseMove={handleMouseMove}
>
 Hover me
</div>
```

EVENT OBJECT

Handlers receive an event object:

```
function handleClick(event) {
 console.log(event.type)
 console.log(event.target)
 console.log(event.currentTarget)
}

<button onClick={handleClick}>Click</button>
```

React wraps native events in SyntheticEvent.
Same interface across browsers.

PREVENT DEFAULT

Stop default browser behavior:

```
function Form() {
 function handleSubmit(event) {
 event.preventDefault()
```

```
 console.log("Form submitted")
 }

 return (
 <form onSubmit={handleSubmit}>
 <input name="email" />
 <button type="submit">Submit</button>
 </form>
 )
}
```

Link without navigation:

```
function Link() {
 function handleClick(event) {
 event.preventDefault()
 console.log("Link clicked, but no navigation")
 }

 return  <a href="/somewhere" onClick={handleClick}>Click
me</a>
}
```

STOP PROPAGATION

Stop event from bubbling up:

```
function Parent() {
 function handleParentClick() {
 console.log("Parent clicked")
 }

 function handleChildClick(event) {
 event.stopPropagation()
 console.log("Child clicked")
 }
```

```
 return (
  <div onClick={handleParentClick}>
  <button onClick={handleChildClick}>Click me</button>
  </div>
 )
}
```

Without stopPropagation, both handlers fire.
With it, only child handler fires.

PASSING ARGUMENTS

Wrap in arrow function:

```
function List({ items, onItemClick }) {
 return (
 <ul>
 {items.map(item => (
 <li key={item.id}>
 <button onClick={() => onItemClick(item.id)}>
 {item.name}
 </button>
 </li>
 ))}
 </ul>
 )
}
```

Or use bind:

```
<button onClick={handleClick.bind(null, item.id)}>
 {item.name}
</button>
```

With event and arguments:

```
function handleClick(id, event) {
 console.log(id, event.target)
}

<button onClick={(e) => handleClick(item.id, e)}>
 Click
</button>
```

FORM EVENTS

Input change:

```
function Form() {
 let [name, setName] = useState("")

 function handleChange(event) {
 setName(event.target.value)
 }

 return (
 <input value={name} onChange={handleChange} />
 )
}
```

Multiple inputs:

```
function Form() {
 let [form, setForm] = useState({ name: "", email: "" })

 function handleChange(event) {
 let { name, value } = event.target
 setForm({ ...form, [name]: value })
 }
```

```
 return (
  <form>
  <input name="name" value={form.name}
onChange={handleChange} />
  <input name="email" value={form.email}
onChange={handleChange} />
  </form>
 )
}
```

Checkbox:

```
function Checkbox() {
 let [checked, setChecked] = useState(false)

 function handleChange(event) {
 setChecked(event.target.checked)
 }

 return (
  <input type="checkbox" checked={checked}
onChange={handleChange} />
 )
}
```

Select:

```
function Select() {
 let [value, setValue] = useState("option1")

 function handleChange(event) {
 setValue(event.target.value)
 }

 return (
 <select value={value} onChange={handleChange}>
 <option value="option1">Option 1</option>
 <option value="option2">Option 2</option>
```

```
 </select>
 )
}
```

KEYBOARD EVENTS

```
function Input() {
 function handleKeyDown(event) {
 if (event.key === "Enter") {
 console.log("Enter pressed")
 }

 if (event.key === "Escape") {
 console.log("Escape pressed")
 }
 }

 return <input onKeyDown={handleKeyDown} />
}
```

Common key values:
- Enter
- Escape
- Tab
- ArrowUp, ArrowDown, ArrowLeft, ArrowRight
- Backspace
- Delete

FOCUS EVENTS

```
function Input() {
 let [focused, setFocused] = useState(false)
```

```
 return (
 <div className={focused ? "focused" : ""}>
 <input
 onFocus={() => setFocused(true)}
 onBlur={() => setFocused(false)}
 />
 </div>
 )
}
```

MOUSE EVENTS

```
function HoverCard() {
 let [hovered, setHovered] = useState(false)

 return (
 <div
 className={`card ${hovered ? "hovered" : ""}`}
 onMouseEnter={() => setHovered(true)}
 onMouseLeave={() => setHovered(false)}
 >
 Hover over me
 </div>
 )
}
```

DRAG AND DROP

```
function Draggable() {
 function handleDragStart(event) {
 event.dataTransfer.setData("text", event.target.id)
 }

 function handleDragOver(event) {
```

```
 event.preventDefault()
 }

 function handleDrop(event) {
 event.preventDefault()
 let data = event.dataTransfer.getData("text")
 console.log("Dropped:", data)
 }

 return (
 <div>
 <div
 id="drag-item"
 draggable
 onDragStart={handleDragStart}
 >
 Drag me
 </div>
 <div
 onDragOver={handleDragOver}
 onDrop={handleDrop}
 >
 Drop here
 </div>
 </div>
 )
}
```

EVENT HANDLER NAMING

Convention: handle + EventName

```
function Component() {
 function handleClick() { }
 function handleSubmit() { }
 function handleChange() { }
 function handleMouseEnter() { }
```

```
 return (...)
}
```

For props: on + EventName

```
<Button onClick={handleClick} />
<Form onSubmit={handleSubmit} />
<Input onChange={handleChange} />
```

DEBOUNCING EVENTS

Limit how often handler runs:

```
function SearchInput() {
 let [query, setQuery] = useState("")

 function handleChange(event) {
 setQuery(event.target.value)
 }

 useEffect(() => {
 let timer = setTimeout(() => {
 if (query) {
 console.log("Searching for:", query)
 }
 }, 300)

 return () => clearTimeout(timer)
 }, [query])

 return <input value={query} onChange={handleChange} />
}
```

PRACTICE

Handle click events.

Build forms with multiple inputs.

Use keyboard events.

Implement hover effects.

Practice event.preventDefault.

WHAT YOU LEARNED

You learned React event syntax.
You learned common event types.
You learned to prevent default behavior.
You learned to stop propagation.
You learned to pass arguments to handlers.
You learned form event handling.

The next chapter teaches conditional rendering.
BOOK 16: REACT FUNDAMENTALS
Chapter 8: Conditional Rendering

WHAT IS CONDITIONAL RENDERING

Show different UI based on conditions.

If user is logged in, show dashboard.
If not, show login form.

React uses JavaScript conditions for this.

IF STATEMENT

Store elements in variables:

```
function Greeting({ isLoggedIn }) {
 let content

 if (isLoggedIn) {
 content = <h1>Welcome back!</h1>
 } else {
 content = <h1>Please sign in</h1>
 }

 return <div>{content}</div>
}
```

Early return pattern:

```
function UserDashboard({ user }) {
 if (!user) {
 return <p>Please log in to view dashboard</p>
 }

 return (
 <div>
 <h1>Welcome, {user.name}</h1>
 <p>Email: {user.email}</p>
 </div>
 )
}
```

TERNARY OPERATOR

Inline condition in JSX:

```
function Greeting({ isLoggedIn }) {
 return (
 <div>
 {isLoggedIn ? (
 <h1>Welcome back!</h1>
 ) : (
 <h1>Please sign in</h1>
 )}
 </div>
 )
}
```

Short form:

```
function Status({ isActive }) {
 return  <span>{isActive ? "Active" : "Inactive"}</span>
}
```

For attributes:

```
function Button({ isPrimary }) {
 return (
 <button className={isPrimary ? "btn-primary" : "btn-secondary"}
>
 Click me
 </button>
 )
}
```

LOGICAL AND OPERATOR

Show element only if condition is true:

```
function Notification({ count }) {
 return (
 <div>
 {count > 0 && <p>You have {count} messages</p>}
 </div>
 )
}
```

If count > 0 is true, renders the paragraph.
If false, renders nothing.

Warning about 0:

```
function Messages({ count }) {
 return (
 <div>
 {count && <p>You have messages</p>}
 </div>
 )
}
```

If count is 0, React renders 0 (not nothing).

Fix:

```
{count > 0 && <p>You have messages</p>}
```

Multiple conditions:

```
function AdminPanel({ user }) {
 return (
 <div>
 {user && user.isAdmin && (
 <button>Delete All</button>
 )}
 </div>
```

```
 )
}
```

LOGICAL OR OPERATOR

Fallback values:

```
function Greeting({ name }) {
 return <h1>Hello, {name || "Guest"}!</h1>
}
```

Nullish coalescing (better):

```
function Greeting({ name }) {
 return <h1>Hello, {name ?? "Guest"}!</h1>
}
```

?? only uses fallback for null or undefined.
|| uses fallback for any falsy value.

SWITCH STATEMENT

Multiple conditions:

```
function StatusBadge({ status }) {
 let badge

 switch (status) {
 case "pending":
 badge = <span className="badge yellow">Pending</span>
 break
 case "approved":
 badge = <span className="badge green">Approved</span>
```

```
 break
 case "rejected":
 badge = <span className="badge red">Rejected</span>
 break
 default:
 badge = <span className="badge gray">Unknown</span>
 }

 return badge
}
```

OBJECT MAPPING

Alternative to switch:

```
function StatusBadge({ status }) {
 let badges = {
 pending: <span className="badge yellow">Pending</span>,
 approved: <span className="badge green">Approved</span>,
 rejected: <span className="badge red">Rejected</span>
 }

 return badges[status] || <span className="badge
gray">Unknown</span>
}
```

For components:

```
function Page({ page }) {
 let pages = {
 home: <HomePage />,
 about: <AboutPage />,
 contact: <ContactPage />
 }

 return pages[page] || <NotFoundPage />
```

```
}
```

CONDITIONAL CLASSES

String concatenation:

```
function Button({ isActive, isDisabled }) {
 let className = "btn"

 if (isActive) className += " active"
 if (isDisabled) className += " disabled"

 return <button className={className}>Click</button>
}
```

Template literal:

```
function Button({ isActive }) {
 return (
 <button className={`btn ${isActive ? "active" : ""}`}>
 Click
 </button>
 )
}
```

Array join:

```
function Button({ isActive, isLarge, isPrimary }) {
 let classes = ["btn"]

 if (isActive) classes.push("active")
 if (isLarge) classes.push("large")
 if (isPrimary) classes.push("primary")

 return <button className={classes.join(" ")}>Click</button>
```

```
}
```

CONDITIONAL STYLES

Inline style object:

```
function Box({ isHighlighted }) {
 return (
 <div style={{
 backgroundColor: isHighlighted ? "yellow" : "white",
 padding: "20px"
 }}>
 Content
 </div>
 )
}
```

Spread conditional styles:

```
function Box({ isHighlighted }) {
 let baseStyles = { padding: "20px" }
 let highlightStyles = isHighlighted ? { backgroundColor: "yellow" }
: {}

 return <div style={{ ...baseStyles, ...highlightStyles }}
>Content</div>
}
```

PREVENTING RENDER

Return null to render nothing:

```
function Warning({ show, message }) {
```

```
 if (!show) {
 return null
 }

 return <div className="warning">{message}</div>
}
```

Component still exists, just renders nothing.
Lifecycle hooks still run.

CONDITIONAL RENDERING PATTERNS

Loading state:

```
function DataDisplay({ loading, error, data }) {
 if (loading) {
 return <p>Loading...</p>
 }

 if (error) {
 return <p>Error: {error.message}</p>
 }

 if (!data) {
 return <p>No data available</p>
 }

 return (
 <div>
 <h1>{data.title}</h1>
 <p>{data.content}</p>
 </div>
 )
}
```

Authentication:

```
function App() {
 let [user, setUser] = useState(null)

 return (
 <div>
 {user ? (
 <Dashboard user={user} onLogout={() => setUser(null)} />
 ) : (
 <LoginForm onLogin={setUser} />
 )}
 </div>
 )
}
```

Feature flags:

```
function App({ features }) {
 return (
 <div>
 <Header />
 {features.newDashboard && <NewDashboard />}
 {!features.newDashboard && <OldDashboard />}
 <Footer />
 </div>
 )
}
```

EXTRACTING CONDITIONAL LOGIC

Move logic to separate function:

```
function getStatusMessage(status) {
 if (status === "loading") return "Loading..."
 if (status === "error") return "Something went wrong"
```

```
 if (status === "success") return "Data loaded successfully"
 return "Unknown status"
}

function Status({ status }) {
 return <p>{getStatusMessage(status)}</p>
}
```

Or to a component:

```
function StatusMessage({ status }) {
 if (status === "loading") return <LoadingSpinner />
 if (status === "error") return <ErrorMessage />
 if (status === "success") return <SuccessMessage />
 return null
}

function App() {
 let [status, setStatus] = useState("loading")

 return (
 <div>
 <StatusMessage status={status} />
 </div>
 )
}
```

PRACTICE

Use if statements for conditions.

Practice ternary operator.

Use logical AND for optional elements.

Build loading/error/success states.

Create conditional classes and styles.

WHAT YOU LEARNED

You learned if statements in React.
You learned ternary operator.
You learned logical operators.
You learned switch and object mapping.
You learned conditional classes.
You learned common patterns.

The next chapter teaches lists and keys.
BOOK 16: REACT FUNDAMENTALS
Chapter 9: Lists and Keys

RENDERING LISTS

Use map() to transform arrays into elements:

```
function NumberList() {
 let numbers = [1, 2, 3, 4, 5]

 return (
 <ul>
 {numbers.map(number => (
 <li key={number}>{number}</li>
 ))}
 </ul>
 )
}
```

With objects:

```
function UserList() {
 let users = [
 { id: 1, name: "Alice" },
 { id: 2, name: "Bob" },
 { id: 3, name: "Charlie" }
 ]

 return (
 <ul>
 {users.map(user => (
 <li key={user.id}>{user.name}</li>
 ))}
 </ul>
 )
}
```

WHY KEYS MATTER

Keys help React identify which items changed.

Without keys, React cannot efficiently update lists.

React uses keys to:
- Track items across re-renders
- Maintain component state
- Optimize DOM updates

KEY RULES

1. Keys must be unique among siblings:

```
<ul>
```

```
 <li key="1">Item 1</li>
 <li key="2">Item 2</li>
 <li key="3">Item 3</li>
</ul>
```

2. Keys should be stable (not change):

Good:
```
{users.map(user => <User key={user.id} />)}
```

Bad:
```
{users.map(user => <User key={Math.random()} />)}
```

3. Don't use array index if list can reorder:

Bad (if items can be added, removed, or reordered):
```
{users.map((user, index) => <User key={index} />)}
```

Good:
```
{users.map(user => <User key={user.id} />)}
```

WHEN INDEX IS OK

Using index as key is acceptable when:
- List is static (never changes)
- Items are never reordered
- Items are never filtered
- Items have no unique ID

```
function StaticList() {
 let items = ["Red", "Green", "Blue"]

 return (
 <ul>
 {items.map((item, index) => (
```

```
 <li key={index}>{item}</li>
 ))}
 </ul>
 )
}
```

GENERATING UNIQUE IDS

If data has no ID, create one:

```
let idCounter = 0

function createId() {
 return `id-${++idCounter}`
}
```

Or use uuid:

```
npm install uuid

import { v4 as uuid } from "uuid"

function addItem(text) {
 setItems([...items, { id: uuid(), text }])
}
```

Or use Date.now():

```
function addItem(text) {
 setItems([...items, { id: Date.now(), text }])
}
```

EXTRACTING LIST ITEMS

Move list items to component:

```
function UserList({ users }) {
 return (
 <ul>
 {users.map(user => (
 <UserItem key={user.id} user={user} />
 ))}
 </ul>
 )
}

function UserItem({ user }) {
 return (
 <li>
 <h3>{user.name}</h3>
 <p>{user.email}</p>
 </li>
 )
}
```

Key goes on the component, not inside it.

KEYS AND FRAGMENTS

When returning multiple elements:

```
function Glossary({ items }) {
 return (
 <dl>
 {items.map(item => (
 <Fragment key={item.id}>
 <dt>{item.term}</dt>
 <dd>{item.definition}</dd>
 </Fragment>
```

```
))}
</dl>
)
}
```

Short syntax `<></>` cannot have keys.
Use `<Fragment key={...}>` instead.

FILTERED LISTS

Filter before mapping:

```
function ActiveUsers({ users }) {
 let activeUsers = users.filter(user => user.active)

 return (
 <ul>
 {activeUsers.map(user => (
 <li key={user.id}>{user.name}</li>
 ))}
 </ul>
 )
}
```

Or inline:

```
function ActiveUsers({ users }) {
 return (
 <ul>
 {users
 .filter(user => user.active)
 .map(user => (
 <li key={user.id}>{user.name}</li>
 ))
 }
 </ul>
```

```
 )
}
```

SORTED LISTS

Sort before mapping:

```
function SortedUsers({ users }) {
 let sorted = [...users].sort((a, b) =>
a.name.localeCompare(b.name))

 return (
 <ul>
 {sorted.map(user => (
 <li key={user.id}>{user.name}</li>
 ))}
 </ul>
 )
}
```

Important: Create new array with [...users] before sorting.
Don't mutate the original.

NESTED LISTS

```
function CategoryList({ categories }) {
 return (
 <div>
 {categories.map(category => (
 <div key={category.id}>
 <h2>{category.name}</h2>
 <ul>
 {category.items.map(item => (
```

```
 <li key={item.id}>{item.name}</li>
 ))}
 </ul>
 </div>
 ))}
 </div>
 )
}
```

Each level needs its own keys.
Keys only need to be unique among siblings.

EMPTY LISTS

Handle empty state:

```
function UserList({ users }) {
 if (users.length === 0) {
 return <p>No users found</p>
 }

 return (
 <ul>
 {users.map(user => (
 <li key={user.id}>{user.name}</li>
 ))}
 </ul>
 )
}
```

LIST WITH ACTIONS

```
function TodoList() {
 let [todos, setTodos] = useState([
```

```
{ id: 1, text: "Learn React", done: false },
{ id: 2, text: "Build app", done: false }
])

function toggleTodo(id) {
setTodos(todos.map(todo =>
todo.id === id ? { ...todo, done: !todo.done } : todo
))
}

function deleteTodo(id) {
setTodos(todos.filter(todo => todo.id !== id))
}

return (
<ul>
{todos.map(todo => (
<li key={todo.id}>
<input
type="checkbox"
checked={todo.done}
onChange={() => toggleTodo(todo.id)}
/>
<span style={{
textDecoration: todo.done ? "line-through" : "none"
}}>
{todo.text}
</span>
<button onClick={() => deleteTodo(todo.id)}>
Delete
</button>
</li>
))}
</ul>
)
}
```

VIRTUALIZED LISTS

For very long lists, render only visible items:

npm install react-window

```
import { FixedSizeList } from "react-window"

function VirtualList({ items }) {
 function Row({ index, style }) {
 return (
 <div style={style}>
 {items[index].name}
 </div>
 )
 }

 return (
 <FixedSizeList
 height={400}
 width={300}
 itemCount={items.length}
 itemSize={35}
 >
 {Row}
 </FixedSizeList>
 )
}
```

Only renders visible items plus buffer.
Essential for thousands of items.

PRACTICE

Render arrays with map.

Use proper keys.

Filter and sort lists.

Handle empty states.

Build interactive lists.

WHAT YOU LEARNED

You learned to render lists.
You learned why keys matter.
You learned key best practices.
You learned to filter and sort.
You learned nested lists.
You learned virtualization.

The next chapter teaches forms and inputs.
BOOK 16: REACT FUNDAMENTALS
Chapter 10: Forms and Inputs

CONTROLLED COMPONENTS

React controls form input values through state:

```
function TextInput() {
 let [value, setValue] = useState("")

 return (
 <input
 type="text"
 value={value}
 onChange={(e) => setValue(e.target.value)}
 />
```

```
 )
}
```

value={value} - Input displays state value
onChange - Updates state when user types

This is a controlled component.
React is the "single source of truth".

UNCONTROLLED COMPONENTS

Let DOM handle input value:

```
import { useRef } from "react"

function TextInput() {
 let inputRef = useRef(null)

 function handleSubmit() {
 console.log(inputRef.current.value)
 }

 return (
 <>
 <input type="text" ref={inputRef} defaultValue="" />
 <button onClick={handleSubmit}>Submit</button>
 </>
 )
}
```

defaultValue sets initial value.
Read value with ref when needed.

Use controlled for most cases.
Use uncontrolled for file inputs or integration with non-React code.

TEXT INPUT

```
function NameInput() {
 let [name, setName] = useState("")

 return (
 <div>
 <label htmlFor="name">Name:</label>
 <input
 id="name"
 type="text"
 value={name}
 onChange={(e) => setName(e.target.value)}
 placeholder="Enter your name"
 />
 <p>Hello, {name || "stranger"}!</p>
 </div>
 )
}
```

TEXTAREA

In React, textarea uses value prop:

```
function MessageInput() {
 let [message, setMessage] = useState("")

 return (
 <textarea
 value={message}
 onChange={(e) => setMessage(e.target.value)}
 rows={4}
 placeholder="Enter message..."
 />
 )
```

}

Unlike HTML where content goes between tags.

SELECT

```
function FruitSelect() {
 let [fruit, setFruit] = useState("apple")

 return (
 <select value={fruit} onChange={(e) =>
setFruit(e.target.value)}>
 <option value="apple">Apple</option>
 <option value="banana">Banana</option>
 <option value="cherry">Cherry</option>
 </select>
 )
}
```

Multiple select:

```
function FruitMultiSelect() {
 let [fruits, setFruits] = useState([])

 function handleChange(e) {
 let options = e.target.options
 let selected = []

 for (let i = 0; i < options.length; i++) {
 if (options[i].selected) {
 selected.push(options[i].value)
 }
 }

 setFruits(selected)
 }
```

```
 return (
 <select multiple value={fruits} onChange={handleChange}>
 <option value="apple">Apple</option>
 <option value="banana">Banana</option>
 <option value="cherry">Cherry</option>
 </select>
 )
}
```

CHECKBOX

Single checkbox:

```
function Newsletter() {
 let [subscribed, setSubscribed] = useState(false)

 return (
 <label>
 <input
 type="checkbox"
 checked={subscribed}
 onChange={(e) => setSubscribed(e.target.checked)}
 />
 Subscribe to newsletter
 </label>
 )
}
```

Multiple checkboxes:

```
function Interests() {
 let [interests, setInterests] = useState({
 sports: false,
 music: false,
 movies: false
```

```
})

function handleChange(e) {
setInterests({
...interests,
[e.target.name]: e.target.checked
})
}

return (
<div>
<label>
<input
type="checkbox"
name="sports"
checked={interests.sports}
onChange={handleChange}
/>
Sports
</label>
<label>
<input
type="checkbox"
name="music"
checked={interests.music}
onChange={handleChange}
/>
Music
</label>
<label>
<input
type="checkbox"
name="movies"
checked={interests.movies}
onChange={handleChange}
/>
Movies
</label>
</div>
)
```

```
}
```

RADIO BUTTONS

```
function GenderSelect() {
 let [gender, setGender] = useState("")

 return (
 <div>
 <label>
 <input
 type="radio"
 name="gender"
 value="male"
 checked={gender === "male"}
 onChange={(e) => setGender(e.target.value)}
 />
 Male
 </label>
 <label>
 <input
 type="radio"
 name="gender"
 value="female"
 checked={gender === "female"}
 onChange={(e) => setGender(e.target.value)}
 />
 Female
 </label>
 <label>
 <input
 type="radio"
 name="gender"
 value="other"
 checked={gender === "other"}
 onChange={(e) => setGender(e.target.value)}
 />
```

```
 Other
 </label>
 </div>
 )
}
```

FILE INPUT

File inputs are always uncontrolled:

```
function FileUpload() {
 let [file, setFile] = useState(null)

 function handleChange(e) {
 setFile(e.target.files[0])
 }

 return (
 <div>
 <input type="file" onChange={handleChange} />
 {file && <p>Selected: {file.name}</p>}
 </div>
 )
}
```

Multiple files:

```
function MultipleFiles() {
 let [files, setFiles] = useState([])

 function handleChange(e) {
 setFiles(Array.from(e.target.files))
 }

 return (
 <div>
```

```
 <input type="file" multiple onChange={handleChange} />
 <ul>
 {files.map((file, i) => (
 <li key={i}>{file.name}</li>
 ))}
 </ul>
 </div>
 )
}
```

COMPLETE FORM

```
function RegistrationForm() {
 let [form, setForm] = useState({
 name: "",
 email: "",
 password: "",
 age: "",
 gender: "",
 newsletter: false
 })

 function handleChange(e) {
 let { name, value, type, checked } = e.target
 setForm({
 ...form,
 [name]: type === "checkbox" ? checked : value
 })
 }

 function handleSubmit(e) {
 e.preventDefault()
 console.log("Form submitted:", form)
 }

 return (
 <form onSubmit={handleSubmit}>
```

```
<div>
<label htmlFor="name">Name:</label>
<input
id="name"
name="name"
type="text"
value={form.name}
onChange={handleChange}
/>
</div>

<div>
<label htmlFor="email">Email:</label>
<input
id="email"
name="email"
type="email"
value={form.email}
onChange={handleChange}
/>
</div>

<div>
<label htmlFor="password">Password:</label>
<input
id="password"
name="password"
type="password"
value={form.password}
onChange={handleChange}
/>
</div>

<div>
<label htmlFor="age">Age:</label>
<input
id="age"
name="age"
type="number"
value={form.age}
```

```
onChange={handleChange}
/>
</div>

<div>
<label>
<input
type="radio"
name="gender"
value="male"
checked={form.gender === "male"}
onChange={handleChange}
/>
Male
</label>
<label>
<input
type="radio"
name="gender"
value="female"
checked={form.gender === "female"}
onChange={handleChange}
/>
Female
</label>
</div>

<div>
<label>
<input
type="checkbox"
name="newsletter"
checked={form.newsletter}
onChange={handleChange}
/>
Subscribe to newsletter
</label>
</div>

<button type="submit">Register</button>
```

```
 </form>
 )
}
```

FORM VALIDATION

```
function ValidatedForm() {
 let [email, setEmail] = useState("")
 let [error, setError] = useState("")

 function validateEmail(value) {
 if (!value) {
 return "Email is required"
 }
 if (!/^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(value)) {
 return "Invalid email format"
 }
 return ""
 }

 function handleChange(e) {
 let value = e.target.value
 setEmail(value)
 setError(validateEmail(value))
 }

 function handleSubmit(e) {
 e.preventDefault()
 let validationError = validateEmail(email)

 if (validationError) {
 setError(validationError)
 return
 }

 console.log("Submitting:", email)
 }
```

```
 return (
 <form onSubmit={handleSubmit}>
 <input
 type="email"
 value={email}
 onChange={handleChange}
 className={error ? "error" : ""}
 />
 {error && <span className="error-message">{error}</span>}
 <button type="submit">Submit</button>
 </form>
 )
}
```

FORM RESET

```
function ResettableForm() {
 let initialState = { name: "", email: "" }
 let [form, setForm] = useState(initialState)

 function handleReset() {
 setForm(initialState)
 }

 return (
 <form>
 <input
 name="name"
 value={form.name}
 onChange={(e) => setForm({ ...form, name: e.target.value })}
 />
 <input
 name="email"
 value={form.email}
 onChange={(e) => setForm({ ...form, email: e.target.value })}
 />
```

```
 <button type="button" onClick={handleReset}>Reset</
button>
 </form>
 )
}
```

PRACTICE

Build controlled text inputs.

Create forms with multiple input types.

Handle checkboxes and radio buttons.

Implement form validation.

Practice form submission.

WHAT YOU LEARNED

You learned controlled components.
You learned all input types.
You learned to handle forms.
You learned validation patterns.
You learned form submission.

The next chapter teaches the useEffect hook.
BOOK 16: REACT FUNDAMENTALS
Chapter 11: useEffect Hook

WHAT IS USEEFFECT

useEffect handles side effects in components.

Side effects are:
- Data fetching
- Subscriptions
- Manually changing DOM
- Timers
- Logging

These don't fit in the render flow.

BASIC USEEFFECT

```
import { useState, useEffect } from "react"

function Counter() {
 let [count, setCount] = useState(0)

 useEffect(() => {
 document.title = `Count: ${count}`
 })

 return (
 <button onClick={() => setCount(count + 1)}>
 Count: {count}
 </button>
 )
}
```

Effect runs after every render.
Updates document title with current count.

USEEFFECT SYNTAX

useEffect(effectFunction, dependencyArray)

effectFunction - Function to run
dependencyArray - Values that trigger re-run (optional)

DEPENDENCY ARRAY

Control when effect runs:

No array - runs after every render:

```
useEffect(() => {
 console.log("Runs after every render")
})
```

Empty array - runs once on mount:

```
useEffect(() => {
 console.log("Runs only once")
}, [])
```

With dependencies - runs when dependencies change:

```
useEffect(() => {
 console.log("Count changed:", count)
}, [count])
```

Multiple dependencies:

```
useEffect(() => {
 console.log("Name or age changed")
}, [name, age])
```

CLEANUP FUNCTION

Return a function to clean up:

```
useEffect(() => {
 let timer = setInterval(() => {
 console.log("Tick")
 }, 1000)

 return () => {
 clearInterval(timer)
 }
}, [])
```

Cleanup runs:
- Before effect runs again
- When component unmounts

Why cleanup matters:

Without cleanup, timers keep running.
Without cleanup, subscriptions stay active.
Without cleanup, memory leaks happen.

DATA FETCHING

```
function UserProfile({ userId }) {
 let [user, setUser] = useState(null)
 let [loading, setLoading] = useState(true)
 let [error, setError] = useState(null)

 useEffect(() => {
```

```
 setLoading(true)
 setError(null)

 fetch(`/api/users/${userId}`)
 .then(res => {
 if (!res.ok) throw new Error("Failed to fetch")
 return res.json()
 })
 .then(data => {
 setUser(data)
 setLoading(false)
 })
 .catch(err => {
 setError(err.message)
 setLoading(false)
 })
 }, [userId])

 if (loading) return <p>Loading...</p>
 if (error) return <p>Error: {error}</p>
 if (!user) return null

 return (
 <div>
 <h1>{user.name}</h1>
 <p>{user.email}</p>
 </div>
 )
}
```

Effect re-runs when userId changes.

ASYNC USEEFFECT

useEffect cannot be async directly.

Wrong:

```
useEffect(async () = > {
 let data = await fetchData()
}, [])
```

Correct (async function inside):

```
useEffect(() = > {
 async function fetchData() {
 let res = await fetch("/api/data")
 let data = await res.json()
 setData(data)
 }

 fetchData()
}, [])
```

Or IIFE:

```
useEffect(() = > {
 (async () = > {
 let res = await fetch("/api/data")
 let data = await res.json()
 setData(data)
 })()
}, [])
```

RACE CONDITIONS

Handle component unmount during fetch:

```
useEffect(() = > {
 let cancelled = false

 async function fetchData() {
```

```
 let res = await fetch(`/api/users/${userId}`)
 let data = await res.json()

 if (!cancelled) {
 setUser(data)
 }
 }

 fetchData()

 return () => {
 cancelled = true
 }
}, [userId])
```

Using AbortController:

```
useEffect(() => {
 let controller = new AbortController()

 async function fetchData() {
 try {
 let res = await fetch(`/api/users/${userId}`, {
 signal: controller.signal
 })
 let data = await res.json()
 setUser(data)
 } catch (err) {
 if (err.name !== "AbortError") {
 setError(err.message)
 }
 }
 }

 fetchData()

 return () => controller.abort()
}, [userId])
```

EVENT LISTENERS

```
function WindowSize() {
 let [size, setSize] = useState({
 width: window.innerWidth,
 height: window.innerHeight
 })

 useEffect(() => {
 function handleResize() {
 setSize({
 width: window.innerWidth,
 height: window.innerHeight
 })
 }

 window.addEventListener("resize", handleResize)

 return () => {
 window.removeEventListener("resize", handleResize)
 }
 }, [])

 return <p>{size.width} x {size.height}</p>
}
```

TIMERS

```
function Timer() {
 let [seconds, setSeconds] = useState(0)

 useEffect(() => {
 let interval = setInterval(() => {
 setSeconds(s => s + 1)
```

```
 }, 1000)

 return () => clearInterval(interval)
 }, [])

 return <p>Seconds: {seconds}</p>
}
```

Countdown:

```
function Countdown({ from }) {
 let [count, setCount] = useState(from)

 useEffect(() => {
 if (count <= 0) return

 let timer = setTimeout(() => {
 setCount(count - 1)
 }, 1000)

 return () => clearTimeout(timer)
 }, [count])

 return <p>{count > 0 ? count : "Done!"}</p>
}
```

LOCAL STORAGE

```
function PersistentCounter() {
 let [count, setCount] = useState(() => {
 let saved = localStorage.getItem("count")
 return saved ? parseInt(saved) : 0
 })

 useEffect(() => {
 localStorage.setItem("count", count)
```

```
 }, [count])

 return (
 <button onClick={() => setCount(count + 1)}>
 Count: {count}
 </button>
 )
}
```

DOCUMENT TITLE

```
function useDocumentTitle(title) {
 useEffect(() => {
 document.title = title
 }, [title])
}

function Page({ title }) {
 useDocumentTitle(title)

 return <h1>{title}</h1>
}
```

MULTIPLE EFFECTS

Separate concerns into multiple effects:

```
function Component({ userId }) {
 useEffect(() => {
 console.log("Component mounted")
 return () => console.log("Component unmounted")
 }, [])

 useEffect(() => {
```

```
 console.log("User changed:", userId)
 }, [userId])

 useEffect(() => {
 let interval = setInterval(() => {
 console.log("Heartbeat")
 }, 5000)

 return () => clearInterval(interval)
 }, [])

 return <div>...</div>
}
```

Each effect handles one concern.

COMMON MISTAKES

1. Missing dependencies:

```
let [count, setCount] = useState(0)

useEffect(() => {
 let interval = setInterval(() => {
 setCount(count + 1)
 }, 1000)

 return () => clearInterval(interval)
}, [])
```

count is stale. Fix with callback:

```
setCount(c => c + 1)
```

2. Object/array dependencies:

```
useEffect(() => {
 console.log("User changed")
}, [{ name: "Alice" }])
```

New object every render. Effect runs every time.

Fix: Use primitive values or useMemo.

PRACTICE

Fetch data with useEffect.

Set up and clean up timers.

Add event listeners.

Sync with localStorage.

Handle race conditions.

WHAT YOU LEARNED

You learned what side effects are.
You learned useEffect syntax.
You learned dependency arrays.
You learned cleanup functions.
You learned data fetching.
You learned common patterns.

The next chapter teaches custom hooks.

Chapter 12: Custom Hooks

WHAT ARE CUSTOM HOOKS

Custom hooks extract reusable logic from components.

They are functions that use other hooks.
Name must start with "use".

Custom hooks let you:
- Share logic between components
- Keep components simple
- Create abstractions

CREATING A CUSTOM HOOK

Before (logic in component):

```
function Counter() {
 let [count, setCount] = useState(0)

 function increment() {
 setCount(count + 1)
 }

 function decrement() {
 setCount(count - 1)
 }

 function reset() {
 setCount(0)
 }

 return (...)
}
```

After (logic in hook):

```
function useCounter(initialValue = 0) {
 let [count, setCount] = useState(initialValue)

 function increment() {
 setCount(c => c + 1)
 }

 function decrement() {
 setCount(c => c - 1)
 }

 function reset() {
 setCount(initialValue)
 }

 return { count, increment, decrement, reset }
}

function Counter() {
 let { count, increment, decrement, reset } = useCounter(0)

 return (
 <div>
 <p>Count: {count}</p>
 <button onClick={increment}>+</button>
 <button onClick={decrement}>-</button>
 <button onClick={reset}>Reset</button>
 </div>
 )
}
```

HOOK RULES

1. Only call hooks at top level.

Not inside loops, conditions, or nested functions.

2. Only call hooks from React functions.
Components or custom hooks.

3. Custom hook names must start with "use".
useCounter, useFetch, useLocalStorage.

USETOGGLE HOOK

```
function useToggle(initialValue = false) {
 let [value, setValue] = useState(initialValue)

 function toggle() {
 setValue(v => !v)
 }

 function setTrue() {
 setValue(true)
 }

 function setFalse() {
 setValue(false)
 }

 return { value, toggle, setTrue, setFalse }
}

function Modal() {
 let { value: isOpen, toggle, setFalse: close } = useToggle(false)

 return (
 <div>
 <button onClick={toggle}>Toggle Modal</button>
 {isOpen && (
 <div className="modal">
 <p>Modal content</p>
```

```
 <button onClick={close}>Close</button>
 </div>
 )}
 </div>
 )
}
```

USEFETCH HOOK

```
function useFetch(url) {
 let [data, setData] = useState(null)
 let [loading, setLoading] = useState(true)
 let [error, setError] = useState(null)

 useEffect(() => {
 let cancelled = false

 async function fetchData() {
 setLoading(true)
 setError(null)

 try {
 let res = await fetch(url)

 if (!res.ok) {
 throw new Error(`HTTP error: ${res.status}`)
 }

 let json = await res.json()

 if (!cancelled) {
 setData(json)
 }
 } catch (err) {
 if (!cancelled) {
 setError(err.message)
 }
```

```
 } finally {
 if (!cancelled) {
 setLoading(false)
 }
 }
 }

 fetchData()

 return () => {
 cancelled = true
 }
 }, [url])

 return { data, loading, error }
}

function UserList() {
 let { data: users, loading, error } = useFetch("/api/users")

 if (loading) return <p>Loading...</p>
 if (error) return <p>Error: {error}</p>

 return (
 <ul>
 {users.map(user => (
 <li key={user.id}>{user.name}</li>
 ))}
 </ul>
 )
}
```

USELOCALSTORAGE HOOK

```
function useLocalStorage(key, initialValue) {
 let [value, setValue] = useState(() => {
 try {
```

```
 let item = localStorage.getItem(key)
 return item ? JSON.parse(item) : initialValue
 } catch {
 return initialValue
 }
 })

 useEffect(() => {
 try {
 localStorage.setItem(key, JSON.stringify(value))
 } catch (error) {
 console.error("Error saving to localStorage:", error)
 }
 }, [key, value])

 return [value, setValue]
}

function App() {
 let [theme, setTheme] = useLocalStorage("theme", "light")

 return (
  <div className={theme}>
  <button onClick={() => setTheme(theme === "light" ? "dark"
: "light")}>
 Toggle Theme
  </button>
  </div>
 )
}
```

USEFORM HOOK

```
function useForm(initialValues) {
 let [values, setValues] = useState(initialValues)
 let [errors, setErrors] = useState({})
 let [touched, setTouched] = useState({})
```

```
function handleChange(e) {
let { name, value, type, checked } = e.target
setValues({
...values,
[name]: type === "checkbox" ? checked : value
})
}

function handleBlur(e) {
let { name } = e.target
setTouched({ ...touched, [name]: true })
}

function reset() {
setValues(initialValues)
setErrors({})
setTouched({})
}

function setFieldValue(name, value) {
setValues({ ...values, [name]: value })
}

function setFieldError(name, error) {
setErrors({ ...errors, [name]: error })
}

return {
values,
errors,
touched,
handleChange,
handleBlur,
reset,
setFieldValue,
setFieldError,
setValues,
setErrors
}
```

```
}

function LoginForm() {
 let { values, handleChange, reset } = useForm({
 email: "",
 password: ""
 })

 function handleSubmit(e) {
 e.preventDefault()
 console.log(values)
 }

 return (
 <form onSubmit={handleSubmit}>
 <input
 name="email"
 value={values.email}
 onChange={handleChange}
 />
 <input
 name="password"
 type="password"
 value={values.password}
 onChange={handleChange}
 />
 <button type="submit">Login</button>
 <button type="button" onClick={reset}>Reset</button>
 </form>
 )
}
```

USEWINDOWSIZE HOOK

```
function useWindowSize() {
 let [size, setSize] = useState({
 width: window.innerWidth,
```

```
 height: window.innerHeight
 })

 useEffect(() => {
 function handleResize() {
 setSize({
 width: window.innerWidth,
 height: window.innerHeight
 })
 }

 window.addEventListener("resize", handleResize)
 return () => window.removeEventListener("resize", handleResize)
 }, [])

 return size
}

function ResponsiveComponent() {
 let { width } = useWindowSize()

 return (
 <div>
 {width < 768 ? <MobileView /> : <DesktopView />}
 </div>
 )
}
```

USEDEBOUNCE HOOK

```
function useDebounce(value, delay) {
 let [debouncedValue, setDebouncedValue] = useState(value)

 useEffect(() => {
 let timer = setTimeout(() => {
 setDebouncedValue(value)
 }, delay)
```

```
 return () => clearTimeout(timer)
 }, [value, delay])

 return debouncedValue
}

function Search() {
 let [query, setQuery] = useState("")
 let debouncedQuery = useDebounce(query, 300)

 useEffect(() => {
 if (debouncedQuery) {
 console.log("Searching for:", debouncedQuery)
 }
 }, [debouncedQuery])

 return (
 <input
 value={query}
 onChange={(e) => setQuery(e.target.value)}
 placeholder="Search..."
 />
 )
}
```

USEPREVIOUS HOOK

```
function usePrevious(value) {
 let ref = useRef()

 useEffect(() => {
 ref.current = value
 }, [value])

 return ref.current
}
```

```
function Counter() {
 let [count, setCount] = useState(0)
 let previousCount = usePrevious(count)

 return (
 <div>
 <p>Current: {count}</p>
 <p>Previous: {previousCount}</p>
 <button onClick={() => setCount(count + 1)}>Increment</
button>
 </div>
 )
}
```

USEONCLICKOUTSIDE HOOK

```
function useOnClickOutside(ref, handler) {
 useEffect(() => {
 function listener(event) {
 if (!ref.current || ref.current.contains(event.target)) {
 return
 }
 handler(event)
 }

 document.addEventListener("mousedown", listener)
 document.addEventListener("touchstart", listener)

 return () => {
 document.removeEventListener("mousedown", listener)
 document.removeEventListener("touchstart", listener)
 }
 }, [ref, handler])
}

function Dropdown() {
```

```
 let [isOpen, setIsOpen] = useState(false)
 let dropdownRef = useRef()

 useOnClickOutside(dropdownRef, () => setIsOpen(false))

 return (
 <div ref={dropdownRef}>
 <button onClick={() => setIsOpen(!isOpen)}>Menu</
button>
 {isOpen && (
 <ul>
 <li>Option 1</li>
 <li>Option 2</li>
 </ul>
 )}
 </div>
 )
}
```

ORGANIZING HOOKS

Create hooks folder:

```
src/
 hooks/
 useCounter.js
 useFetch.js
 useLocalStorage.js
 useForm.js
 index.js
```

Export from index.js:

```
export { useCounter } from "./useCounter"
export { useFetch } from "./useFetch"
export { useLocalStorage } from "./useLocalStorage"
```

export { useForm } from "./useForm"

Import:

import { useFetch, useLocalStorage } from "@/hooks"

PRACTICE

Create useToggle hook.

Create useFetch hook.

Create useLocalStorage hook.

Create useDebounce hook.

Organize hooks in folder.

WHAT YOU LEARNED

You learned what custom hooks are.
You learned hook rules.
You learned to extract logic.
You learned common hook patterns.
You learned to organize hooks.

The next chapter teaches the useRef hook.
BOOK 16: REACT FUNDAMENTALS
Chapter 13: useRef Hook

WHAT IS USEREF

useRef creates a mutable reference that persists across renders.

Two main uses:
- Access DOM elements directly
- Store values that don't trigger re-renders

```
import { useRef } from "react"

let ref = useRef(initialValue)
```

ref.current contains the value.

ACCESSING DOM ELEMENTS

```
function TextInput() {
 let inputRef = useRef(null)

 function focusInput() {
 inputRef.current.focus()
 }

 return (
 <div>
 <input ref={inputRef} type="text" />
 <button onClick={focusInput}>Focus Input</button>
 </div>
 )
}
```

ref={inputRef} attaches ref to element.
inputRef.current is the actual DOM element.

COMMON DOM OPERATIONS

Focus element:

```
function FocusInput() {
 let inputRef = useRef(null)

 useEffect(() => {
 inputRef.current.focus()
 }, [])

 return <input ref={inputRef} />
}
```

Scroll into view:

```
function ScrollToElement() {
 let elementRef = useRef(null)

 function scrollTo() {
 elementRef.current.scrollIntoView({ behavior: "smooth" })
 }

 return (
 <div>
 <button onClick={scrollTo}>Scroll to element</button>
 <div style={{ height: "2000px" }} />
 <div ref={elementRef}>Target element</div>
 </div>
 )
}
```

Measure element:

```
function MeasuredBox() {
 let boxRef = useRef(null)
 let [dimensions, setDimensions] = useState({ width: 0, height: 0 })
```

```
 useEffect(() => {
 if (boxRef.current) {
 let { width, height } = boxRef.current.getBoundingClientRect()
 setDimensions({ width, height })
 }
 }, [])

 return (
 <div>
 <div ref={boxRef} style={{ padding: "20px", border: "1px solid
black" }}>
 Content here
 </div>
 <p>Width: {dimensions.width}, Height: {dimensions.height}</
p>
 </div>
 )
}
```

VIDEO AND AUDIO CONTROL

```
function VideoPlayer() {
 let videoRef = useRef(null)

 function play() {
 videoRef.current.play()
 }

 function pause() {
 videoRef.current.pause()
 }

 function toggleMute() {
 videoRef.current.muted = !videoRef.current.muted
 }
```

```
 return (
 <div>
 <video ref={videoRef} src="/video.mp4" width="400" />
 <div>
 <button onClick={play}>Play</button>
 <button onClick={pause}>Pause</button>
 <button onClick={toggleMute}>Toggle Mute</button>
 </div>
 </div>
 )
}
```

STORING VALUES WITHOUT RE-RENDER

Values that persist but don't trigger re-render:

```
function Timer() {
 let [count, setCount] = useState(0)
 let intervalRef = useRef(null)

 function start() {
 if (intervalRef.current) return

 intervalRef.current = setInterval(() => {
 setCount(c => c + 1)
 }, 1000)
 }

 function stop() {
 clearInterval(intervalRef.current)
 intervalRef.current = null
 }

 useEffect(() => {
 return () => clearInterval(intervalRef.current)
 }, [])
```

```
 return (
 <div>
 <p>Count: {count}</p>
 <button onClick={start}>Start</button>
 <button onClick={stop}>Stop</button>
 </div>
 )
}
```

intervalRef doesn't cause re-render when changed.

STORING PREVIOUS VALUE

```
function PreviousValue() {
 let [count, setCount] = useState(0)
 let previousRef = useRef()

 useEffect(() => {
 previousRef.current = count
 }, [count])

 return (
 <div>
 <p>Current: {count}</p>
 <p>Previous: {previousRef.current}</p>
 <button onClick={() => setCount(count + 1)}>Increment</
button>
 </div>
 )
}
```

USEREF VS USESTATE

useState:

- Triggers re-render when changed
- Use for values that affect UI

useRef:
- Does NOT trigger re-render
- Use for values that don't affect UI

When to use useRef:
- DOM references
- Timer IDs
- Previous values
- Any mutable value not affecting render

When to use useState:
- Data displayed in UI
- Form values
- Loading states
- Any value that should update UI

CALLBACK REFS

For more control over when ref is attached:

```
function MeasuredComponent() {
 let [height, setHeight] = useState(0)

 let measuredRef = useCallback(node => {
 if (node !== null) {
 setHeight(node.getBoundingClientRect().height)
 }
 }, [])

 return (
 <div>
 <div ref={measuredRef}>Content with variable height</div>
 <p>Height: {height}px</p>
```

```
 </div>
 )
}
```

Callback runs when ref is attached or detached.

FORWARDING REFS

Pass ref to child component:

```
import { forwardRef } from "react"

let FancyInput = forwardRef(function FancyInput(props, ref) {
 return <input ref={ref} className="fancy-input" {...props} />
})

function Parent() {
 let inputRef = useRef(null)

 function focusInput() {
 inputRef.current.focus()
 }

 return (
 <div>
 <FancyInput ref={inputRef} placeholder="Enter text" />
 <button onClick={focusInput}>Focus</button>
 </div>
 )
}
```

forwardRef allows parent to access child's DOM element.

MULTIPLE REFS

Use object or array:

```
function MultipleInputs() {
 let inputRefs = useRef({})

 function focusInput(name) {
 inputRefs.current[name]?.focus()
 }

 return (
 <div>
 <input
 ref={el => inputRefs.current.name = el}
 placeholder="Name"
 />
 <input
 ref={el => inputRefs.current.email = el}
 placeholder="Email"
 />
 <button onClick={() => focusInput("name")}>Focus Name</
button>
 <button onClick={() => focusInput("email")}>Focus Email</
button>
 </div>
 )
}
```

With array:

```
function DynamicInputs({ count }) {
 let inputRefs = useRef([])

 return (
 <div>
 {Array.from({ length: count }).map((_, i) => (
 <input
 key={i}
 ref={el => inputRefs.current[i] = el}
```

```
 placeholder={`Input ${i + 1}`}
 />
 ))}
 <button onClick={() => inputRefs.current[0]?.focus()}>
 Focus First
 </button>
 </div>
 )
}
```

UNCONTROLLED INPUTS

Read value on submit instead of tracking every change:

```
function UncontrolledForm() {
 let nameRef = useRef(null)
 let emailRef = useRef(null)

 function handleSubmit(e) {
 e.preventDefault()
 console.log({
 name: nameRef.current.value,
 email: emailRef.current.value
 })
 }

 return (
 <form onSubmit={handleSubmit}>
 <input ref={nameRef} name="name" defaultValue="" />
 <input ref={emailRef} name="email" defaultValue="" />
 <button type="submit">Submit</button>
 </form>
 )
}
```

CANVAS DRAWING

```
function Canvas() {
 let canvasRef = useRef(null)

 useEffect(() => {
 let canvas = canvasRef.current
 let ctx = canvas.getContext("2d")

 ctx.fillStyle = "blue"
 ctx.fillRect(10, 10, 100, 100)

 ctx.strokeStyle = "red"
 ctx.beginPath()
 ctx.arc(200, 60, 50, 0, Math.PI * 2)
 ctx.stroke()
 }, [])

 return <canvas ref={canvasRef} width={400} height={200} />
}
```

PRACTICE

Focus input on mount.

Control video playback.

Store timer reference.

Use forwardRef.

Build uncontrolled form.

WHAT YOU LEARNED

You learned to access DOM elements.
You learned to store mutable values.
You learned ref vs state.
You learned callback refs.
You learned to forward refs.

The next chapter teaches useContext hook.

BOOK 16: REACT FUNDAMENTALS

Chapter 14: useContext Hook

THE PROBLEM: PROP DRILLING

Passing props through many levels:

```
function App() {
 let [user, setUser] = useState({ name: "Alice" })

 return <Layout user={user} />
}

function Layout({ user }) {
 return <Main user={user} />
}

function Main({ user }) {
 return <Sidebar user={user} />
}

function Sidebar({ user }) {
 return <UserInfo user={user} />
}

function UserInfo({ user }) {
 return <p>Hello, {user.name}</p>
```

}

user passes through Layout, Main, Sidebar just to reach UserInfo.
This is prop drilling. It's messy.

WHAT IS CONTEXT

Context provides data to entire component tree.

No need to pass props through every level.

Components can access context directly.

CREATING CONTEXT

```
import { createContext, useContext, useState } from "react"

let UserContext = createContext(null)

function App() {
 let [user, setUser] = useState({ name: "Alice" })

 return (
 <UserContext.Provider value={user}>
 <Layout />
 </UserContext.Provider>
 )
}

function Layout() {
 return <Main />
}

function Main() {
```

```
 return <Sidebar />
}

function Sidebar() {
 return <UserInfo />
}

function UserInfo() {
 let user = useContext(UserContext)

 return <p>Hello, {user.name}</p>
}
```

No prop drilling. UserInfo accesses context directly.

CONTEXT API

Create context:

```
let MyContext = createContext(defaultValue)
```

Provide context (wrap tree):

```
<MyContext.Provider value={someValue}>
 <ChildComponents />
</MyContext.Provider>
```

Consume context:

```
let value = useContext(MyContext)
```

CONTEXT WITH STATE AND FUNCTIONS

Provide both data and functions:

```
let AuthContext = createContext(null)

function AuthProvider({ children }) {
 let [user, setUser] = useState(null)

 function login(userData) {
 setUser(userData)
 }

 function logout() {
 setUser(null)
 }

 let value = { user, login, logout }

 return (
 <AuthContext.Provider value={value}>
 {children}
 </AuthContext.Provider>
 )
}

function useAuth() {
 let context = useContext(AuthContext)

 if (!context) {
 throw new Error("useAuth must be used within AuthProvider")
 }

 return context
}

function App() {
 return (
 <AuthProvider>
 <Header />
 <Main />
```

```
 </AuthProvider>
 )
}

function Header() {
 let { user, logout } = useAuth()

 return (
 <header>
 {user ? (
 <>
 <span>Hello, {user.name}</span>
 <button onClick={logout}>Logout</button>
 </>
 ) : (
 <span>Please log in</span>
 )}
 </header>
 )
}

function LoginForm() {
 let { login } = useAuth()

 function handleSubmit(e) {
 e.preventDefault()
 login({ name: "Alice" })
 }

 return (
 <form onSubmit={handleSubmit}>
 <button type="submit">Login</button>
 </form>
 )
}
```

THEME CONTEXT

```
let ThemeContext = createContext("light")

function ThemeProvider({ children }) {
 let [theme, setTheme] = useState("light")

 function toggleTheme() {
 setTheme(t => t === "light" ? "dark" : "light")
 }

 return (
 <ThemeContext.Provider value={{ theme, toggleTheme }}>
 {children}
 </ThemeContext.Provider>
 )
}

function useTheme() {
 return useContext(ThemeContext)
}

function App() {
 return (
 <ThemeProvider>
 <Layout />
 </ThemeProvider>
 )
}

function Layout() {
 let { theme } = useTheme()

 return (
 <div className={`app ${theme}`}>
 <ThemeToggle />
 <Content />
 </div>
 )
}
```

```
function ThemeToggle() {
 let { theme, toggleTheme } = useTheme()

 return (
 <button onClick={toggleTheme}>
 Current: {theme}. Click to toggle.
 </button>
 )
}
```

MULTIPLE CONTEXTS

Combine multiple contexts:

```
function App() {
 return (
 <AuthProvider>
 <ThemeProvider>
 <LanguageProvider>
 <Layout />
 </LanguageProvider>
 </ThemeProvider>
 </AuthProvider>
 )
}
```

Or combine into one:

```
let AppContext = createContext(null)

function AppProvider({ children }) {
 let [user, setUser] = useState(null)
 let [theme, setTheme] = useState("light")
 let [language, setLanguage] = useState("en")

 let value = {
```

```
 user, setUser,
 theme, setTheme,
 language, setLanguage
 }

 return (
 <AppContext.Provider value={value}>
 {children}
 </AppContext.Provider>
 )
}
```

CONTEXT WITH REDUCER

For complex state:

```
import { createContext, useContext, useReducer } from "react"

let CartContext = createContext(null)

function cartReducer(state, action) {
 switch (action.type) {
 case "ADD_ITEM":
 return { ...state, items: [...state.items, action.payload] }

 case "REMOVE_ITEM":
 return {
 ...state,
 items: state.items.filter(item => item.id !== action.payload)
 }

 case "CLEAR_CART":
 return { ...state, items: [] }

 default:
 return state
 }
```

```
}

function CartProvider({ children }) {
 let [state, dispatch] = useReducer(cartReducer, { items: [] })

 function addItem(item) {
 dispatch({ type: "ADD_ITEM", payload: item })
 }

 function removeItem(id) {
 dispatch({ type: "REMOVE_ITEM", payload: id })
 }

 function clearCart() {
 dispatch({ type: "CLEAR_CART" })
 }

 let value = { items: state.items, addItem, removeItem, clearCart }

 return (
 <CartContext.Provider value={value}>
 {children}
 </CartContext.Provider>
 )
}

function useCart() {
 let context = useContext(CartContext)

 if (!context) {
 throw new Error("useCart must be used within CartProvider")
 }

 return context
}
```

AVOIDING RE-RENDERS

Split context to avoid unnecessary re-renders:

```
let UserContext = createContext(null)
let UserActionsContext = createContext(null)

function UserProvider({ children }) {
 let [user, setUser] = useState(null)

 let actions = useMemo(() => ({
 login: (data) => setUser(data),
 logout: () => setUser(null)
 }), [])

 return (
 <UserContext.Provider value={user}>
 <UserActionsContext.Provider value={actions}>
 {children}
 </UserActionsContext.Provider>
 </UserContext.Provider>
 )
}

function useUser() {
 return useContext(UserContext)
}

function useUserActions() {
 return useContext(UserActionsContext)
}
```

Components using only actions won't re-render when user changes.

DEFAULT VALUE

Context default value used when no Provider:

```
let ThemeContext = createContext("light")

function Component() {
 let theme = useContext(ThemeContext)

 return <div className={theme}>...</div>
}
```

If no ThemeProvider wraps Component, theme is "light".

WHEN TO USE CONTEXT

Good use cases:
- Theme (dark/light mode)
- Current user authentication
- Preferred language
- UI state (sidebar open/closed)

Avoid for:
- Frequently changing data (use state management)
- Data that only some components need (use props)

PRACTICE

Create AuthContext with login/logout.

Create ThemeContext with toggle.

Use context with reducer.

Split context to optimize.

Build complete provider pattern.

WHAT YOU LEARNED

You learned what prop drilling is.
You learned to create context.
You learned Provider pattern.
You learned context with state.
You learned context with reducer.
You learned optimization patterns.

The next chapter teaches component lifecycle.BOOK 16: REACT FUNDAMENTALS
Chapter 15: Component Lifecycle

WHAT IS LIFECYCLE

Components go through phases:
- Mounting (component appears)
- Updating (component changes)
- Unmounting (component disappears)

Understanding lifecycle helps manage side effects.

LIFECYCLE WITH USEEFFECT

Function components use useEffect for lifecycle:

Mounting (componentDidMount):

```
useEffect(() => {
 console.log("Component mounted")
```

```
}, [])
```

Empty dependency array runs once on mount.

Unmounting (componentWillUnmount):

```
useEffect(() => {
 console.log("Component mounted")

 return () => {
 console.log("Component will unmount")
 }
}, [])
```

Return function runs on unmount.

Updating (componentDidUpdate):

```
useEffect(() => {
 console.log("Count changed to:", count)
}, [count])
```

Runs when count changes.

COMPLETE LIFECYCLE EXAMPLE

```
function LifecycleDemo() {
 let [count, setCount] = useState(0)

 useEffect(() => {
 console.log("1. Component mounted")

 return () => {
 console.log("4. Component unmounting")
 }
```

```
 }, [])

 useEffect(() => {
 console.log("2. Component rendered")
 })

 useEffect(() => {
 console.log("3. Count updated:", count)
 }, [count])

 return (
 <div>
 <p>Count: {count}</p>
 <button onClick={() => setCount(count + 1)}>Increment</
button>
 </div>
 )
}
```

MOUNTING PHASE

When component first appears in DOM:

```
function MountingExample() {
 let [data, setData] = useState(null)
 let [loading, setLoading] = useState(true)

 useEffect(() => {
 console.log("Component mounted - fetching data")

 async function fetchData() {
 let response = await fetch("/api/data")
 let json = await response.json()
 setData(json)
 setLoading(false)
 }
```

```
 fetchData()
 }, [])

 if (loading) return <p>Loading...</p>

 return <div>{data?.title}</div>
}
```

Common mount actions:
- Fetch initial data
- Set up subscriptions
- Initialize third-party libraries
- Focus an input

UPDATING PHASE

When props or state change:

```
function UpdatingExample({ userId }) {
 let [user, setUser] = useState(null)

 useEffect(() => {
 console.log("userId changed - fetching user", userId)

 async function fetchUser() {
 let response = await fetch(`/api/users/${userId}`)
 let json = await response.json()
 setUser(json)
 }

 fetchUser()
 }, [userId])

 useEffect(() => {
 if (user) {
 console.log("user data changed:", user.name)
 }
```

```
 }, [user])

 return <div>{user?.name}</div>
}
```

Common update actions:
- Fetch new data when ID changes
- Update document title
- Sync with external systems
- Log analytics

UNMOUNTING PHASE

When component is removed:

```
function UnmountingExample() {
 useEffect(() => {
 let interval = setInterval(() => {
 console.log("Tick")
 }, 1000)

 let handleResize = () => console.log("Resized")
 window.addEventListener("resize", handleResize)

 return () => {
 console.log("Cleaning up...")
 clearInterval(interval)
 window.removeEventListener("resize", handleResize)
 }
 }, [])

 return <div>Timer running...</div>
}
```

Common cleanup actions:
- Clear timers (clearInterval, clearTimeout)
- Remove event listeners

- Cancel network requests
- Unsubscribe from subscriptions
- Close WebSocket connections

CONDITIONAL RENDERING AND LIFECYCLE

```
function App() {
 let [showComponent, setShowComponent] = useState(false)

 return (
 <div>
 <button onClick={() => setShowComponent(!
showComponent)}>
 Toggle Component
 </button>
 {showComponent && <ChildComponent />}
 </div>
 )
}

function ChildComponent() {
 useEffect(() => {
 console.log("ChildComponent mounted")

 return () => {
 console.log("ChildComponent unmounted")
 }
 }, [])

 return <p>I am the child component</p>
}
```

Toggle causes mount/unmount cycle.

DEPENDENCIES AND RE-RUNS

Effect runs when dependencies change:

```
function DependencyExample({ a, b }) {
 useEffect(() => {
 console.log("Effect runs - a or b changed")
 }, [a, b])

 useEffect(() => {
 console.log("Effect runs - only when a changes")
 }, [a])

 useEffect(() => {
 console.log("Effect runs - every render")
 })

 useEffect(() => {
 console.log("Effect runs - only once on mount")
 }, [])

 return <div>a: {a}, b: {b}</div>
}
```

CLEANUP BEFORE RE-RUN

Cleanup runs before effect re-runs:

```
function ChatRoom({ roomId }) {
 useEffect(() => {
 let connection = connectToRoom(roomId)
 console.log(`Connected to room ${roomId}`)

 return () => {
 connection.disconnect()
 console.log(`Disconnected from room ${roomId}`)
 }
```

```
 }, [roomId])

 return <div>Chat Room: {roomId}</div>
}
```

When roomId changes:
1. Cleanup for old roomId runs
2. Effect for new roomId runs

EFFECT TIMING

Effects run after render:

```
function TimingExample() {
 let [count, setCount] = useState(0)

 console.log("1. Component rendering, count:", count)

 useEffect(() => {
 console.log("3. Effect running, count:", count)
 })

 console.log("2. Before return")

 return (
 <button onClick={() => setCount(count + 1)}>
 Count: {count}
 </button>
 )
}
```

Output order:
1. Component rendering
2. Before return
3. Effect running (after DOM updated)

USELAYOUTEFFECT

Runs synchronously after DOM mutations:

```
import { useLayoutEffect } from "react"

function LayoutEffectExample() {
 let ref = useRef(null)

 useLayoutEffect(() => {
 let { height } = ref.current.getBoundingClientRect()
 console.log("Height:", height)
 })

 return <div ref={ref}>Measured content</div>
}
```

useLayoutEffect vs useEffect:
- useLayoutEffect: runs before browser paint
- useEffect: runs after browser paint

Use useLayoutEffect for:
- DOM measurements
- Preventing flicker
- Synchronous DOM mutations

STRICT MODE AND DOUBLE EFFECTS

In development with StrictMode, effects run twice:

```
<React.StrictMode>
 <App />
</React.StrictMode>

function Component() {
```

```
 useEffect(() => {
 console.log("Mount")
 return () => console.log("Unmount")
 }, [])
}
```

In development, logs:
Mount
Unmount
Mount

This helps catch bugs. Only happens in development.

COMMON PATTERNS

Loading state:

```
function DataComponent() {
 let [data, setData] = useState(null)
 let [loading, setLoading] = useState(true)
 let [error, setError] = useState(null)

 useEffect(() => {
 let mounted = true

 async function fetchData() {
 try {
 let response = await fetch("/api/data")
 let json = await response.json()

 if (mounted) {
 setData(json)
 setLoading(false)
 }
 } catch (err) {
 if (mounted) {
 setError(err.message)
```

```
 setLoading(false)
 }
 }
 }

 fetchData()

 return () => {
 mounted = false
 }
 }, [])

 if (loading) return <p>Loading...</p>
 if (error) return <p>Error: {error}</p>

 return <div>{data.title}</div>
}
```

PRACTICE

Use mount effect for data fetching.

Use cleanup for subscriptions.

Handle prop changes with effects.

Use useLayoutEffect for measurements.

Build loading/error/success patterns.

WHAT YOU LEARNED

You learned component lifecycle phases.
You learned mount, update, unmount with useEffect.

You learned cleanup functions.
You learned effect timing.
You learned useLayoutEffect.

The next chapter teaches styling components.
BOOK 16: REACT FUNDAMENTALS
Chapter 16: Styling Components

STYLING OPTIONS

Multiple ways to style React:
- Inline styles
- CSS files
- CSS Modules
- CSS-in-JS
- Utility classes (Tailwind)

Choose based on project needs.

INLINE STYLES

JavaScript objects for styles:

```
function Button() {
 let buttonStyle = {
 backgroundColor: "blue",
 color: "white",
 padding: "10px 20px",
 border: "none",
 borderRadius: "4px",
 cursor: "pointer"
 }
```

```
 return <button style={buttonStyle}>Click me</button>
}
```

Properties use camelCase:

```
let style = {
 backgroundColor: "red",
 fontSize: "16px",
 marginTop: "20px",
 paddingLeft: "10px"
}
```

Dynamic inline styles:

```
function Box({ isActive }) {
 let style = {
 backgroundColor: isActive ? "green" : "gray",
 padding: "20px"
 }

 return <div style={style}>Box content</div>
}
```

CSS FILES

Import CSS directly:

src/components/Button.css:

```
.button {
 background-color: blue;
 color: white;
 padding: 10px 20px;
 border: none;
 border-radius: 4px;
```

```
 cursor: pointer;
}

.button:hover {
 background-color: darkblue;
}

.button.primary {
 background-color: #007bff;
}

.button.secondary {
 background-color: #6c757d;
}
```

src/components/Button.jsx:

```
import "./Button.css"

function Button({ variant = "primary", children }) {
 return (
 <button className={`button ${variant}`}>
 {children}
 </button>
 )
}
```

Problem: CSS is global. Class names can conflict.

CSS MODULES

Scoped CSS with unique class names:

src/components/Button.module.css:

```
.button {
 background-color: blue;
 color: white;
 padding: 10px 20px;
 border: none;
 border-radius: 4px;
}

.primary {
 background-color: #007bff;
}

.secondary {
 background-color: #6c757d;
}
```

src/components/Button.jsx:

```
import styles from "./Button.module.css"

function Button({ variant = "primary", children }) {
 return (
 <button className={`${styles.button} ${styles[variant]}`}>
 {children}
 </button>
 )
}
```

styles.button becomes unique like Button_button_a1b2c.
No class name conflicts.

DYNAMIC CLASSES

With template literals:

```
function Card({ isActive, isHighlighted }) {
```

```
 let className = `card ${isActive ? "active" : ""} ${isHighlighted ? "highlighted" : ""}`

 return <div className={className.trim()}>Content</div>
}
```

With array join:

```
function Card({ isActive, isHighlighted }) {
 let classes = ["card"]

 if (isActive) classes.push("active")
 if (isHighlighted) classes.push("highlighted")

 return <div className={classes.join(" ")}>Content</div>
}
```

With classnames library:

```
npm install classnames

import cn from "classnames"

function Card({ isActive, isHighlighted }) {
 return (
 <div className={cn("card", {
 active: isActive,
 highlighted: isHighlighted
 })}>
 Content
 </div>
 )
}
```

STYLED COMPONENTS

CSS-in-JS library:

npm install styled-components

```
import styled from "styled-components"

let Button = styled.button`
 background-color: ${props => props.primary ? "#007bff" :
"#6c757d"};
 color: white;
 padding: 10px 20px;
 border: none;
 border-radius: 4px;
 cursor: pointer;

 &:hover {
 opacity: 0.9;
 }
`

function App() {
 return (
 <div>
 <Button primary>Primary</Button>
 <Button>Secondary</Button>
 </div>
 )
}
```

Extending styles:

```
let BaseButton = styled.button`
 padding: 10px 20px;
 border: none;
 border-radius: 4px;
`

let PrimaryButton = styled(BaseButton)`
```

```
 background-color: #007bff;
 color: white;
`

let DangerButton = styled(BaseButton)`
 background-color: #dc3545;
 color: white;
`
```

EMOTION

Another CSS-in-JS library:

```
npm install @emotion/react @emotion/styled

import styled from "@emotion/styled"
import { css } from "@emotion/react"

let Button = styled.button`
 background-color: blue;
 color: white;
 padding: 10px 20px;
`

let dynamicStyle = props => css`
 color: ${props.color};
 font-size: ${props.size}px;
`

function App() {
 return (
 <div css={dynamicStyle({ color: "red", size: 20 })}>
 Styled text
 </div>
 )
}
```

TAILWIND CSS

Utility-first CSS framework:

```
npm install tailwindcss postcss autoprefixer
npx tailwindcss init -p
```

tailwind.config.js:

```
module.exports = {
 content: ["./src/**/*.{js,jsx,ts,tsx}"],
 theme: {
 extend: {}
 },
 plugins: []
}
```

src/index.css:

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

Using Tailwind:

```
function Button({ variant }) {
 let baseClasses = "px-4 py-2 rounded font-medium"
 let variantClasses = variant === "primary"
 ? "bg-blue-500 text-white hover:bg-blue-600"
 : "bg-gray-500 text-white hover:bg-gray-600"

 return (
 <button className={`${baseClasses} ${variantClasses}`}>
 Click me
```

```
 </button>
 )
}
```

Component with Tailwind:

```
function Card({ title, children }) {
 return (
 <div className="bg-white rounded-lg shadow-md p-6 mb-4">
 <h2 className="text-xl font-bold mb-2">{title}</h2>
 <div className="text-gray-600">{children}</div>
 </div>
 )
}
```

GLOBAL STYLES

index.css for global styles:

```
* {
 margin: 0;
 padding: 0;
 box-sizing: border-box;
}

body {
 font-family: -apple-system, BlinkMacSystemFont, "Segoe UI",
Roboto, sans-serif;
 line-height: 1.5;
 color: #333;
}

a {
 color: inherit;
 text-decoration: none;
}
```

Import in main.jsx:

```
import "./index.css"
```

CSS VARIABLES

Define in CSS:

```
:root {
 --primary-color: #007bff;
 --secondary-color: #6c757d;
 --spacing-sm: 8px;
 --spacing-md: 16px;
 --spacing-lg: 24px;
}

.button {
 background-color: var(--primary-color);
 padding: var(--spacing-sm) var(--spacing-md);
}
```

Change with JavaScript:

```
function ThemeProvider({ theme, children }) {
 useEffect(() = > {
 document.documentElement.style.setProperty(
 "--primary-color",
 theme = = = "dark" ? "#4a90d9" : "#007bff"
 )
 }, [theme])

 return children
}
```

RESPONSIVE STYLES

Media queries in CSS:

```
.container {
 padding: 20px;
}

@media (min-width: 768px) {
 .container {
 padding: 40px;
 max-width: 720px;
 margin: 0 auto;
 }
}

@media (min-width: 1024px) {
 .container {
 max-width: 960px;
 }
}
```

With Tailwind:

```
<div className="p-4 md:p-8 lg:p-12 max-w-full md:max-w-3xl
lg:max-w-5xl mx-auto">
 Content
</div>
```

ANIMATIONS

CSS animations:

```
@keyframes fadeIn {
 from { opacity: 0; }
 to { opacity: 1; }
}

.fade-in {
 animation: fadeIn 0.3s ease-in;
}

.slide-in {
 animation: slideIn 0.3s ease-out;
}

@keyframes slideIn {
 from { transform: translateY(-20px); opacity: 0; }
 to { transform: translateY(0); opacity: 1; }
}
```

Conditional animation:

```
function Modal({ isOpen }) {
 if (!isOpen) return null

 return (
 <div className="modal fade-in">
 <div className="modal-content slide-in">
 Modal content
 </div>
 </div>
 )
}
```

PRACTICE

Use CSS Modules.

Try styled-components.

Set up Tailwind CSS.

Create dynamic classes.

Build responsive layouts.

WHAT YOU LEARNED

You learned inline styles.
You learned CSS Modules.
You learned CSS-in-JS.
You learned Tailwind CSS.
You learned dynamic styling.
You learned responsive patterns.

The next chapter teaches React Router.

BOOK 16: REACT FUNDAMENTALS

Chapter 17: React Router

WHAT IS ROUTING

Single-page applications need navigation.

Different URLs show different content.
Browser back/forward buttons work.
URLs can be shared and bookmarked.

React Router handles this.

INSTALLING REACT ROUTER

npm install react-router-dom

Basic setup:

```
import { BrowserRouter, Routes, Route } from "react-router-dom"

function App() {
 return (
 <BrowserRouter>
 <Routes>
 <Route path="/" element={<Home />} />
 <Route path="/about" element={<About />} />
 <Route path="/contact" element={<Contact />} />
 </Routes>
 </BrowserRouter>
 )
}

function Home() {
 return <h1>Home Page</h1>
}

function About() {
 return <h1>About Page</h1>
}

function Contact() {
 return <h1>Contact Page</h1>
}
```

LINK COMPONENT

Navigate without page reload:

```
import { Link } from "react-router-dom"

function Navigation() {
 return (
 <nav>
 <Link to="/">Home</Link>
 <Link to="/about">About</Link>
 <Link to="/contact">Contact</Link>
 </nav>
 )
}

function App() {
 return (
 <BrowserRouter>
 <Navigation />
 <Routes>
 <Route path="/" element={<Home />} />
 <Route path="/about" element={<About />} />
 <Route path="/contact" element={<Contact />} />
 </Routes>
 </BrowserRouter>
 )
}
```

NAVLINK COMPONENT

Link with active styling:

```
import { NavLink } from "react-router-dom"

function Navigation() {
 return (
 <nav>
 <NavLink
 to="/"
```

```
 className={({ isActive }) => isActive ? "active" : ""}
 >
 Home
 </NavLink>
 <NavLink
 to="/about"
 className={({ isActive }) => isActive ? "active" : ""}
 >
 About
 </NavLink>
 </nav>
 )
}
```

Or with style:

```
 <NavLink
 to="/about"
 style={({ isActive }) => ({
 color: isActive ? "red" : "black",
 fontWeight: isActive ? "bold" : "normal"
 })}
 >
 About
 </NavLink>
```

URL PARAMETERS

Dynamic route segments:

```
import { useParams } from "react-router-dom"

function App() {
 return (
 <BrowserRouter>
 <Routes>
```

```
 <Route path="/users/:userId" element={<UserProfile />} />
 <Route path="/posts/:postId" element={<PostDetail />} />
 </Routes>
 </BrowserRouter>
 )
}

function UserProfile() {
 let { userId } = useParams()

 return <h1>User Profile: {userId}</h1>
}

function PostDetail() {
 let { postId } = useParams()
 let [post, setPost] = useState(null)

 useEffect(() => {
 fetch(`/api/posts/${postId}`)
 .then(res => res.json())
 .then(setPost)
 }, [postId])

 if (!post) return <p>Loading...</p>

 return (
 <div>
 <h1>{post.title}</h1>
 <p>{post.content}</p>
 </div>
 )
}
```

NESTED ROUTES

Routes inside routes:

```
function App() {
 return (
 <BrowserRouter>
 <Routes>
 <Route path="/" element={<Layout />}>
 <Route index element={<Home />} />
 <Route path="about" element={<About />} />
 <Route path="users" element={<Users />}>
 <Route index element={<UserList />} />
 <Route path=":userId" element={<UserProfile />} />
 </Route>
 </Route>
 </Routes>
 </BrowserRouter>
 )
}
```

Layout with Outlet:

```
import { Outlet } from "react-router-dom"

function Layout() {
 return (
 <div>
 <Navigation />
 <main>
 <Outlet />
 </main>
 <Footer />
 </div>
 )
}
```

Outlet renders the matched child route.

PROGRAMMATIC NAVIGATION

Navigate from code:

```
import { useNavigate } from "react-router-dom"

function LoginForm() {
 let navigate = useNavigate()

 async function handleSubmit(e) {
 e.preventDefault()
 await login()
 navigate("/dashboard")
 }

 return (
 <form onSubmit={handleSubmit}>
 <button type="submit">Login</button>
 </form>
 )
}
```

Navigate with options:

```
navigate("/dashboard")
navigate("/dashboard", { replace: true })
navigate(-1)
navigate(1)
```

SEARCH PARAMS

URL query strings:

```
import { useSearchParams } from "react-router-dom"

function ProductList() {
 let [searchParams, setSearchParams] = useSearchParams()
```

```
 let category = searchParams.get("category")
 let sort = searchParams.get("sort")

 function handleCategoryChange(newCategory) {
 setSearchParams({ ...Object.fromEntries(searchParams), category:
newCategory })
 }

 return (
 <div>
 <p>Category: {category}</p>
 <p>Sort: {sort}</p>
 <button onClick={() => handleCategoryChange("electronics")}
>
 Electronics
 </button>
 </div>
 )
}
```

LOCATION AND STATE

Get current location:

```
import { useLocation } from "react-router-dom"

function CurrentPath() {
 let location = useLocation()

 return (
 <div>
 <p>Pathname: {location.pathname}</p>
 <p>Search: {location.search}</p>
 <p>Hash: {location.hash}</p>
 </div>
 )
```

```
}
```

Pass state with navigation:

```
<Link to="/checkout" state={{ from: "cart" }}>
 Checkout
</Link>

navigate("/checkout", { state: { from: "cart" } })
```

Access state:

```
function Checkout() {
 let location = useLocation()
 let from = location.state?.from

 return <p>Came from: {from}</p>
}
```

404 NOT FOUND

Catch unmatched routes:

```
function App() {
 return (
  <BrowserRouter>
  <Routes>
  <Route path="/" element={<Home />} />
  <Route path="/about" element={<About />} />
  <Route path="*" element={<NotFound />} />
  </Routes>
  </BrowserRouter>
 )
}
```

```
function NotFound() {
 return (
 <div>
 <h1>404 - Page Not Found</h1>
 <Link to="/">Go Home</Link>
 </div>
 )
}
```

PROTECTED ROUTES

Require authentication:

```
function ProtectedRoute({ children }) {
 let { user } = useAuth()
 let location = useLocation()

 if (!user) {
 return <Navigate to="/login" state={{ from: location }} replace />
 }

 return children
}

function App() {
 return (
 <BrowserRouter>
 <Routes>
 <Route path="/login" element={<Login />} />
 <Route
 path="/dashboard"
 element={
 <ProtectedRoute>
 <Dashboard />
 </ProtectedRoute>
 }
```

```
/>
</Routes>
</BrowserRouter>
)
}
```

Redirect after login:

```
function Login() {
 let navigate = useNavigate()
 let location = useLocation()
 let from = location.state?.from?.pathname || "/"

 async function handleLogin() {
 await login()
 navigate(from, { replace: true })
 }

 return <button onClick={handleLogin}>Login</button>
}
```

LAZY LOADING ROUTES

Load route components on demand:

```
import { lazy, Suspense } from "react"

let Dashboard = lazy(() => import("./pages/Dashboard"))
let Settings = lazy(() => import("./pages/Settings"))

function App() {
 return (
 <BrowserRouter>
 <Suspense fallback={<p>Loading...</p>}>
 <Routes>
 <Route path="/" element={<Home />} />
```

```
 <Route path="/dashboard" element={<Dashboard />} />
 <Route path="/settings" element={<Settings />} />
 </Routes>
 </Suspense>
 </BrowserRouter>
 )
}
```

COMPLETE ROUTING EXAMPLE

```
import { BrowserRouter, Routes, Route, Link, NavLink, useParams,
useNavigate, Navigate } from "react-router-dom"

function App() {
 return (
 <BrowserRouter>
 <nav>
 <NavLink to="/" end>Home</NavLink>
 <NavLink to="/products">Products</NavLink>
 <NavLink to="/about">About</NavLink>
 </nav>

 <Routes>
 <Route path="/" element={<Home />} />
 <Route path="/products" element={<Products />} />
 <Route path="/products/:id" element={<ProductDetail />} />
 <Route path="/about" element={<About />} />
 <Route path="*" element={<NotFound />} />
 </Routes>
 </BrowserRouter>
 )
}

function Home() {
 return <h1>Welcome</h1>
}
```

```
function Products() {
 let products = [
 { id: 1, name: "Product A" },
 { id: 2, name: "Product B" }
 ]

 return (
 <div>
 <h1>Products</h1>
 <ul>
 {products.map(p => (
 <li key={p.id}>
 <Link to={`/products/${p.id}`}>{p.name}</Link>
 </li>
 ))}
 </ul>
 </div>
 )
}

function ProductDetail() {
 let { id } = useParams()
 let navigate = useNavigate()

 return (
 <div>
 <h1>Product {id}</h1>
 <button onClick={() => navigate("/products")}>Back</
button>
 </div>
 )
}
```

PRACTICE

Set up React Router.

Create navigation with links.

Use URL parameters.

Implement nested routes.

Build protected routes.

WHAT YOU LEARNED

You learned React Router setup.
You learned Link and NavLink.
You learned URL parameters.
You learned nested routes.
You learned programmatic navigation.
You learned protected routes.

The next chapter teaches fetching data.

BOOK 16: REACT FUNDAMENTALS
Chapter 18: Fetching Data

DATA FETCHING IN REACT

React components often need external data.

Common sources:
- REST APIs
- GraphQL
- Backend services

Fetch data in useEffect or use libraries.

BASIC DATA FETCHING

```
function UserList() {
 let [users, setUsers] = useState([])
 let [loading, setLoading] = useState(true)
 let [error, setError] = useState(null)

 useEffect(() => {
 fetch("/api/users")
 .then(res => {
 if (!res.ok) throw new Error("Failed to fetch")
 return res.json()
 })
 .then(data => {
 setUsers(data)
 setLoading(false)
 })
 .catch(err => {
 setError(err.message)
 setLoading(false)
 })
 }, [])

 if (loading) return <p>Loading...</p>
 if (error) return <p>Error: {error}</p>

 return (
 <ul>
 {users.map(user => (
 <li key={user.id}>{user.name}</li>
 ))}
 </ul>
 )
}
```

ASYNC/AWAIT PATTERN

```
function UserList() {
 let [users, setUsers] = useState([])
 let [loading, setLoading] = useState(true)
 let [error, setError] = useState(null)

 useEffect(() => {
 async function fetchUsers() {
 try {
 let response = await fetch("/api/users")

 if (!response.ok) {
 throw new Error("Failed to fetch")
 }

 let data = await response.json()
 setUsers(data)
 } catch (err) {
 setError(err.message)
 } finally {
 setLoading(false)
 }
 }

 fetchUsers()
 }, [])

 if (loading) return <p>Loading...</p>
 if (error) return <p>Error: {error}</p>

 return (
 <ul>
 {users.map(user => (
 <li key={user.id}>{user.name}</li>
 ))}
 </ul>
 )
}
```

CANCELING REQUESTS

Prevent updates on unmounted component:

```
function UserProfile({ userId }) {
 let [user, setUser] = useState(null)
 let [loading, setLoading] = useState(true)

 useEffect(() => {
 let controller = new AbortController()

 async function fetchUser() {
 try {
 let response = await fetch(`/api/users/${userId}`, {
 signal: controller.signal
 })
 let data = await response.json()
 setUser(data)
 } catch (err) {
 if (err.name !== "AbortError") {
 console.error(err)
 }
 } finally {
 setLoading(false)
 }
 }

 fetchUser()

 return () => controller.abort()
 }, [userId])

 if (loading) return <p>Loading...</p>
 if (!user) return <p>User not found</p>

 return <h1>{user.name}</h1>
}
```

CUSTOM FETCH HOOK

```
function useFetch(url) {
 let [data, setData] = useState(null)
 let [loading, setLoading] = useState(true)
 let [error, setError] = useState(null)

 useEffect(() => {
 let controller = new AbortController()

 async function fetchData() {
 setLoading(true)
 setError(null)

 try {
 let response = await fetch(url, {
 signal: controller.signal
 })

 if (!response.ok) {
 throw new Error(`HTTP error: ${response.status}`)
 }

 let json = await response.json()
 setData(json)
 } catch (err) {
 if (err.name !== "AbortError") {
 setError(err.message)
 }
 } finally {
 setLoading(false)
 }
 }

 fetchData()
```

```
 return () => controller.abort()
 }, [url])

 return { data, loading, error }
}

function UserList() {
 let { data: users, loading, error } = useFetch("/api/users")

 if (loading) return <p>Loading...</p>
 if (error) return <p>Error: {error}</p>

 return (
 <ul>
 {users.map(user => (
 <li key={user.id}>{user.name}</li>
 ))}
 </ul>
 )
}
```

POST REQUESTS

```
function CreateUser() {
 let [name, setName] = useState("")
 let [loading, setLoading] = useState(false)
 let [error, setError] = useState(null)

 async function handleSubmit(e) {
 e.preventDefault()
 setLoading(true)
 setError(null)

 try {
 let response = await fetch("/api/users", {
 method: "POST",
 headers: {
```

```
"Content-Type": "application/json"
},
body: JSON.stringify({ name })
})

if (!response.ok) {
throw new Error("Failed to create user")
}

let newUser = await response.json()
console.log("Created:", newUser)
setName("")
} catch (err) {
setError(err.message)
} finally {
setLoading(false)
}
}

return (
<form onSubmit={handleSubmit}>
<input
value={name}
onChange={(e) => setName(e.target.value)}
disabled={loading}
/>
<button type="submit" disabled={loading}>
{loading ? "Creating..." : "Create User"}
</button>
{error && <p className="error">{error}</p>}
</form>
)
}
```

USING AXIOS

npm install axios

```
import axios from "axios"

function UserList() {
 let [users, setUsers] = useState([])
 let [loading, setLoading] = useState(true)
 let [error, setError] = useState(null)

 useEffect(() => {
 let source = axios.CancelToken.source()

 async function fetchUsers() {
 try {
 let response = await axios.get("/api/users", {
 cancelToken: source.token
 })
 setUsers(response.data)
 } catch (err) {
 if (!axios.isCancel(err)) {
 setError(err.message)
 }
 } finally {
 setLoading(false)
 }
 }

 fetchUsers()

 return () => source.cancel()
 }, [])

 if (loading) return <p>Loading...</p>
 if (error) return <p>Error: {error}</p>

 return (
 <ul>
 {users.map(user => (
 <li key={user.id}>{user.name}</li>
 ))}
 </ul>
```

```
 )
}
```

REACT QUERY

Powerful data fetching library:

npm install @tanstack/react-query

Setup:

```
import { QueryClient, QueryClientProvider, useQuery, useMutation
} from "@tanstack/react-query"

let queryClient = new QueryClient()

function App() {
 return (
  <QueryClientProvider client={queryClient}>
  <UserList />
  </QueryClientProvider>
 )
}
```

Fetching with useQuery:

```
function UserList() {
 let { data: users, isLoading, error } = useQuery({
 queryKey: ["users"],
 queryFn: () => fetch("/api/users").then(res => res.json())
 })

 if (isLoading) return <p>Loading...</p>
 if (error) return <p>Error: {error.message}</p>
```

```
 return (
 <ul>
 {users.map(user => (
 <li key={user.id}>{user.name}</li>
 ))}
 </ul>
 )
}
```

With parameters:

```
function UserProfile({ userId }) {
 let { data: user, isLoading } = useQuery({
 queryKey: ["user", userId],
 queryFn: () => fetch(`/api/users/${userId}`).then(res =>
res.json())
 })

 if (isLoading) return <p>Loading...</p>

 return <h1>{user.name}</h1>
}
```

Mutations:

```
function CreateUser() {
 let [name, setName] = useState("")

 let mutation = useMutation({
 mutationFn: (newUser) => {
 return fetch("/api/users", {
 method: "POST",
 headers: { "Content-Type": "application/json" },
 body: JSON.stringify(newUser)
 }).then(res => res.json())
 },
 onSuccess: () => {
 queryClient.invalidateQueries({ queryKey: ["users"] })
```

```
 setName("")
 }
 })

 function handleSubmit(e) {
 e.preventDefault()
 mutation.mutate({ name })
 }

 return (
 <form onSubmit={handleSubmit}>
 <input value={name} onChange={(e) =>
setName(e.target.value)} />
 <button disabled={mutation.isPending}>
 {mutation.isPending ? "Creating..." : "Create"}
 </button>
 </form>
 )
}
```

SWR

Another data fetching library:

```
npm install swr

import useSWR from "swr"

let fetcher = url => fetch(url).then(res => res.json())

function UserList() {
 let { data: users, error, isLoading } = useSWR("/api/users",
fetcher)

 if (isLoading) return <p>Loading...</p>
 if (error) return <p>Error loading users</p>
```

```
 return (
 <ul>
 {users.map(user => (
 <li key={user.id}>{user.name}</li>
 ))}
 </ul>
 )
}
```

LOADING STATES

Skeleton loaders:

```
function UserListSkeleton() {
 return (
 <div className="skeleton-list">
 {[1, 2, 3].map(i => (
 <div key={i} className="skeleton-item" />
 ))}
 </div>
 )
}

function UserList() {
 let { data: users, isLoading } = useFetch("/api/users")

 if (isLoading) return <UserListSkeleton />

 return (
 <ul>
 {users.map(user => (
 <li key={user.id}>{user.name}</li>
 ))}
 </ul>
 )
}
```

PAGINATION

```
function PaginatedList() {
 let [page, setPage] = useState(1)
 let { data, loading } = useFetch(`/api/users?page=${page}
&limit=10`)

 return (
 <div>
 {loading ? (
 <p>Loading...</p>
 ) : (
 <ul>
 {data?.users.map(user => (
 <li key={user.id}>{user.name}</li>
 ))}
 </ul>
 )}

 <div className="pagination">
 <button
 onClick={() => setPage(p => Math.max(1, p - 1))}
 disabled={page === 1}
 >
 Previous
 </button>
 <span>Page {page}</span>
 <button
 onClick={() => setPage(p => p + 1)}
 disabled={data?.users.length < 10}
 >
 Next
 </button>
 </div>
 </div>
 )
}
```

PRACTICE

Fetch data with useEffect.

Create useFetch hook.

Handle loading and errors.

Use React Query.

Implement pagination.

WHAT YOU LEARNED

You learned basic data fetching.
You learned to cancel requests.
You learned custom fetch hooks.
You learned React Query.
You learned loading states.
You learned pagination.

The next chapter teaches error boundaries.
BOOK 16: REACT FUNDAMENTALS
Chapter 19: Error Boundaries

WHAT ARE ERROR BOUNDARIES

JavaScript errors can crash React apps.

Error boundaries:
- Catch errors in child components
- Display fallback UI
- Log errors
- Prevent app crash

They are like try-catch for components.

THE PROBLEM

Without error boundaries:

```
function BrokenComponent() {
 throw new Error("Something went wrong!")
 return  <div>This never renders</div>
}

function App() {
 return (
  <div>
  <Header />
  <BrokenComponent />
  <Footer />
  </div>
 )
}
```

Error in BrokenComponent crashes entire app.
User sees blank screen.

CREATING ERROR BOUNDARY

Error boundaries must be class components:

```
import { Component } from "react"

class ErrorBoundary extends Component {
 constructor(props) {
 super(props)
 this.state = { hasError: false, error: null }
 }

 static getDerivedStateFromError(error) {
 return { hasError: true, error }
 }

 componentDidCatch(error, errorInfo) {
 console.error("Error caught:", error)
 console.error("Error info:", errorInfo)
 }

 render() {
 if (this.state.hasError) {
 return this.props.fallback || <h1>Something went wrong.</h1>
 }

 return this.props.children
 }
}
```

Using it:

```
function App() {
 return (
 <div>
 <Header />
 <ErrorBoundary fallback={<p>Error in main content</p>}>
 <MainContent />
 </ErrorBoundary>
 <Footer />
 </div>
 )
}
```

Error in MainContent shows fallback.
Header and Footer still work.

GRANULAR ERROR BOUNDARIES

Wrap different sections:

```
function App() {
 return (
  <div>
  <ErrorBoundary fallback={<p>Header error</p>}>
  <Header />
  </ErrorBoundary>

  <ErrorBoundary fallback={<p>Sidebar error</p>}>
  <Sidebar />
  </ErrorBoundary>

  <ErrorBoundary fallback={<p>Content error</p>}>
  <MainContent />
  </ErrorBoundary>

  <ErrorBoundary fallback={<p>Footer error</p>}>
  <Footer />
  </ErrorBoundary>
  </div>
 )
}
```

One broken section doesn't break others.

RESETTABLE ERROR BOUNDARY

Allow retry:

```
class ErrorBoundary extends Component {
 constructor(props) {
 super(props)
 this.state = { hasError: false, error: null }
 }

 static getDerivedStateFromError(error) {
 return { hasError: true, error }
 }

 componentDidCatch(error, errorInfo) {
 console.error("Error:", error, errorInfo)
 }

 resetError = () => {
 this.setState({ hasError: false, error: null })
 }

 render() {
 if (this.state.hasError) {
 return (
 <div className="error-fallback">
 <h2>Something went wrong</h2>
 <p>{this.state.error?.message}</p>
 <button onClick={this.resetError}>Try again</button>
 </div>
 )
 }

 return this.props.children
 }
}
```

ERROR BOUNDARY WITH RENDER PROP

Flexible fallback:

```
class ErrorBoundary extends Component {
 constructor(props) {
 super(props)
 this.state = { hasError: false, error: null }
 }

 static getDerivedStateFromError(error) {
 return { hasError: true, error }
 }

 componentDidCatch(error, errorInfo) {
 if (this.props.onError) {
 this.props.onError(error, errorInfo)
 }
 }

 resetError = () => {
 this.setState({ hasError: false, error: null })
 }

 render() {
 if (this.state.hasError) {
 if (this.props.fallbackRender) {
 return this.props.fallbackRender({
 error: this.state.error,
 resetError: this.resetError
 })
 }

 return this.props.fallback || <h1>Something went wrong</h1>
 }

 return this.props.children
 }
}
```

Usage:

```
<ErrorBoundary
fallbackRender={({ error, resetError }) => (
<div>
<p>Error: {error.message}</p>
<button onClick={resetError}>Retry</button>
</div>
)}
onError={(error, info) => logToService(error, info)}
>
<MyComponent />
</ErrorBoundary>
```

REACT-ERROR-BOUNDARY LIBRARY

```
npm install react-error-boundary

import { ErrorBoundary } from "react-error-boundary"

function ErrorFallback({ error, resetErrorBoundary }) {
return (
<div role="alert">
<p>Something went wrong:</p>
<pre>{error.message}</pre>
<button onClick={resetErrorBoundary}>Try again</button>
</div>
)
}

function App() {
return (
<ErrorBoundary
FallbackComponent={ErrorFallback}
onError={(error, info) => console.error(error, info)}
onReset={() => console.log("Reset")}
>
<MyComponent />
```

```
  </ErrorBoundary>
 )
}
```

With reset keys:

```
<ErrorBoundary
 FallbackComponent={ErrorFallback}
 resetKeys={[userId]}
>
 <UserProfile userId={userId} />
</ErrorBoundary>
```

Automatically resets when userId changes.

WHAT ERROR BOUNDARIES CATCH

Error boundaries catch:
- Errors during rendering
- Errors in lifecycle methods
- Errors in constructors

Error boundaries do NOT catch:
- Event handlers
- Async code (setTimeout, promises)
- Server-side rendering
- Errors in error boundary itself

HANDLING EVENT HANDLER ERRORS

Use try-catch in event handlers:

```
function Button() {
```

```
 let [error, setError] = useState(null)

 function handleClick() {
 try {
 doSomethingRisky()
 } catch (err) {
 setError(err.message)
 }
 }

 if (error) {
 return <p>Error: {error}</p>
 }

 return <button onClick={handleClick}>Click me</button>
}
```

HANDLING ASYNC ERRORS

Catch in useEffect or handlers:

```
function DataLoader() {
 let [error, setError] = useState(null)

 useEffect(() => {
 async function fetchData() {
 try {
 let response = await fetch("/api/data")

 if (!response.ok) {
 throw new Error("Failed to fetch")
 }

 let data = await response.json()
 } catch (err) {
 setError(err.message)
 }
```

```
 }

 fetchData()
 }, [])

 if (error) {
 return <p>Error: {error}</p>
 }

 return <div>Data loaded</div>
}
```

ERROR LOGGING

Log errors to external service:

```
class ErrorBoundary extends Component {
 componentDidCatch(error, errorInfo) {
 logErrorToService({
 error: error.message,
 stack: error.stack,
 componentStack: errorInfo.componentStack,
 timestamp: new Date().toISOString()
 })
 }

 render() {
 if (this.state.hasError) {
 return <ErrorFallback />
 }
 return this.props.children
 }
}

function logErrorToService(errorData) {
 fetch("/api/log-error", {
 method: "POST",
```

```
 headers: { "Content-Type": "application/json" },
 body: JSON.stringify(errorData)
 })
}
```

DEVELOPMENT VS PRODUCTION

In development:
- React shows error overlay
- Full error details visible

In production:
- Error boundary catches error
- Fallback UI shown
- No detailed error to users

BEST PRACTICES

1. Wrap at appropriate levels:
 Not too high (everything fails together)
 Not too low (too many boundaries)

2. Provide helpful fallback UI:
 Show what went wrong
 Offer retry option
 Provide navigation

3. Log errors:
 Track errors in production
 Monitor error frequency

4. Handle async errors separately:
 Use try-catch in handlers
 Set error state

PRACTICE

Create ErrorBoundary class.

Wrap components with boundaries.

Add reset functionality.

Use react-error-boundary.

Handle async errors.

WHAT YOU LEARNED

You learned what error boundaries are.
You learned to create error boundaries.
You learned reset patterns.
You learned what they catch and don't catch.
You learned error logging.
You learned best practices.

The next chapter teaches performance optimization.
BOOK 16: REACT FUNDAMENTALS
Chapter 20: Performance Optimization

WHY OPTIMIZE

React is fast by default.

But large apps can become slow:
- Too many re-renders
- Expensive calculations
- Large bundles
- Slow data fetching

Optimization makes apps feel snappy.

REACT DEVTOOLS PROFILER

Measure before optimizing:

1. Install React DevTools browser extension
2. Open Profiler tab
3. Click record
4. Interact with your app
5. Stop recording
6. Analyze results

Profiler shows:
- Which components rendered
- How long each took
- Why they rendered

USEMEMO

Memoize expensive calculations:

```
import { useMemo } from "react"

function ExpensiveComponent({ items, filter }) {
 let filteredItems = useMemo(() => {
 console.log("Filtering items...")
 return items.filter(item => item.name.includes(filter))
```

```
}, [items, filter])

return (
<ul>
{filteredItems.map(item => (
<li key={item.id}>{item.name}</li>
))}
</ul>
)
}
```

Without useMemo: filters on every render.
With useMemo: filters only when items or filter change.

When to use:
- Expensive computations
- Creating objects/arrays passed to child components
- Derived data from props/state

USECALLBACK

Memoize functions:

```
import { useCallback } from "react"

function Parent() {
let [count, setCount] = useState(0)

let handleClick = useCallback(() => {
console.log("Clicked")
}, [])

return (
<div>
<p>Count: {count}</p>
<button onClick={() => setCount(count + 1)}>Increment</
```

```
button>
 <MemoizedChild onClick={handleClick} />
 </div>
 )
}
```

Without useCallback: handleClick is new function every render.
With useCallback: same function reference.

When to use:
- Functions passed to memoized children
- Functions used in useEffect dependencies
- Event handlers that need stable reference

REACT.MEMO

Prevent unnecessary re-renders:

```
import { memo } from "react"

let ExpensiveChild = memo(function ExpensiveChild({ name }) {
 console.log("ExpensiveChild rendering")
 return <div>Hello, {name}</div>
})

function Parent() {
 let [count, setCount] = useState(0)

 return (
 <div>
 <button onClick={() => setCount(count + 1)}>
 Count: {count}
 </button>
 <ExpensiveChild name="Alice" />
 </div>
 )
```

```
}
```

Without memo: ExpensiveChild re-renders when count changes.
With memo: ExpensiveChild only re-renders when name changes.

Custom comparison:

```
let MemoizedComponent = memo(MyComponent, (prevProps,
nextProps) => {
 return prevProps.id === nextProps.id
})
```

COMBINING MEMO AND USECALLBACK

```
function Parent() {
 let [count, setCount] = useState(0)
 let [name, setName] = useState("Alice")

 let handleClick = useCallback(() => {
 console.log("Button clicked")
 }, [])

 let data = useMemo(() => ({
 name,
 timestamp: Date.now()
 }), [name])

 return (
 <div>
 <button onClick={() => setCount(count + 1)}>
 Count: {count}
 </button>
 <MemoizedChild data={data} onClick={handleClick} />
 </div>
 )
}
```

```
let MemoizedChild = memo(function Child({ data, onClick }) {
 console.log("Child rendering")
 return <button onClick={onClick}>{data.name}</button>
})
```

Child only re-renders when name changes, not count.

VIRTUALIZATION

Render only visible items:

```
npm install react-window

import { FixedSizeList } from "react-window"

function VirtualList({ items }) {
 function Row({ index, style }) {
 return (
 <div style={style}>
 {items[index].name}
 </div>
 )
 }

 return (
 <FixedSizeList
 height={400}
 width="100%"
 itemCount={items.length}
 itemSize={50}
 >
 {Row}
 </FixedSizeList>
 )
}
```

10,000 items but only visible ones rendered.

CODE SPLITTING

Split code into smaller bundles:

```
import { lazy, Suspense } from "react"

let Dashboard = lazy(() => import("./Dashboard"))
let Settings = lazy(() => import("./Settings"))

function App() {
 return (
 <Suspense fallback={<p>Loading...</p>}>
 <Routes>
 <Route path="/dashboard" element={<Dashboard />} />
 <Route path="/settings" element={<Settings />} />
 </Routes>
 </Suspense>
 )
}
```

Dashboard and Settings load only when needed.

AVOIDING UNNECESSARY STATE

Wrong (creating state from props):

```
function Component({ initialValue }) {
 let [value, setValue] = useState(initialValue)
}
```

initialValue changes won't update state.

Sometimes you don't need state:

```
function Component({ items }) {
 let total = items.reduce((sum, item) => sum + item.price, 0)

 return <p>Total: {total}</p>
}
```

Derived values don't need state.

BATCHING UPDATES

React automatically batches state updates:

```
function Component() {
 let [a, setA] = useState(0)
 let [b, setB] = useState(0)

 function handleClick() {
 setA(1)
 setB(2)
 }
}
```

Single render, not two.

KEY FOR RESETTING COMPONENTS

Force component to reset:

```
function Parent() {
 let [userId, setUserId] = useState(1)
```

```
 return <UserProfile key={userId} userId={userId} />
}
```

Changing key unmounts and remounts component.
Fresh state for new user.

DEBOUNCING AND THROTTLING

Debounce expensive operations:

```
function Search() {
 let [query, setQuery] = useState("")
 let [results, setResults] = useState([])

 useEffect(() => {
 if (!query) {
 setResults([])
 return
 }

 let timer = setTimeout(async () => {
 let data = await searchAPI(query)
 setResults(data)
 }, 300)

 return () => clearTimeout(timer)
 }, [query])

 return (
 <div>
 <input
 value={query}
 onChange={(e) => setQuery(e.target.value)}
 />
 <ul>
 {results.map(r => <li key={r.id}>{r.name}</li>)}
 </ul>
```

```
 </div>
 )
}
```

AVOIDING INLINE OBJECTS

Bad (new object every render):

```
<Child style={{ color: "red" }} />
<Child config={{ enabled: true }} />
```

Good (stable reference):

```
let style = { color: "red" }
let config = { enabled: true }

function Parent() {
 return (
 <div>
 <Child style={style} />
 <Child config={config} />
 </div>
 )
}
```

Or with useMemo:

```
function Parent({ color }) {
 let style = useMemo(() => ({ color }), [color])

 return <Child style={style} />
}
```

PRODUCTION BUILD

Development is slower than production:

npm run build

Production build:
- Minified code
- No development warnings
- Optimized React

OPTIMIZATION CHECKLIST

1. Profile first
 Don't guess. Measure.

2. Memoize expensive calculations
 useMemo for values.

3. Memoize callbacks
 useCallback for functions.

4. Memoize components
 React.memo for expensive components.

5. Virtualize long lists
 react-window or similar.

6. Code split routes
 lazy() and Suspense.

7. Avoid unnecessary state
 Derive values when possible.

8. Stable references
 Avoid inline objects/functions.

PRACTICE

Use React DevTools Profiler.

Apply useMemo to calculations.

Use useCallback with memo.

Implement virtualized list.

Code split your routes.

WHAT YOU LEARNED

You learned to profile with DevTools.
You learned useMemo.
You learned useCallback.
You learned React.memo.
You learned virtualization.
You learned code splitting.

The next chapter brings everything together with practice programs.

Chapter 21: Practice React Programs

TODO APPLICATION

Complete Todo app with all CRUD operations:

```
import { useState, useEffect } from "react"

function TodoApp() {
 let [todos, setTodos] = useState(() => {
 let saved = localStorage.getItem("todos")
 return saved ? JSON.parse(saved) : []
 })
 let [input, setInput] = useState("")
 let [filter, setFilter] = useState("all")

 useEffect(() => {
 localStorage.setItem("todos", JSON.stringify(todos))
 }, [todos])

 function addTodo(e) {
 e.preventDefault()
 if (!input.trim()) return

 setTodos([
 ...todos,
 {
 id: Date.now(),
 text: input.trim(),
 completed: false,
 createdAt: new Date().toISOString()
 }
 ])
 setInput("")
 }

 function toggleTodo(id) {
 setTodos(todos.map(todo =>
 todo.id === id ? { ...todo, completed: !todo.completed } : todo
 ))
 }

 function deleteTodo(id) {
 setTodos(todos.filter(todo => todo.id !== id))
 }
```

```
function clearCompleted() {
setTodos(todos.filter(todo => !todo.completed))
}

let filteredTodos = todos.filter(todo => {
if (filter === "active") return !todo.completed
if (filter === "completed") return todo.completed
return true
})

let activeCount = todos.filter(t => !t.completed).length

return (
<div className="todo-app">
<h1>Todo List</h1>

<form onSubmit={addTodo}>
<input
value={input}
onChange={(e) => setInput(e.target.value)}
placeholder="What needs to be done?"
/>
<button type="submit">Add</button>
</form>

<div className="filters">
<button
className={filter === "all" ? "active" : ""}
onClick={() => setFilter("all")}
>
All
</button>
<button
className={filter === "active" ? "active" : ""}
onClick={() => setFilter("active")}
>
Active
</button>
<button
```

```
 className={filter === "completed" ? "active" : ""}
 onClick={() => setFilter("completed")}
 >
 Completed
 </button>
 </div>

 <ul className="todo-list">
 {filteredTodos.map(todo => (
 <li key={todo.id} className={todo.completed ? "completed" :
""}>
 <input
 type="checkbox"
 checked={todo.completed}
 onChange={() => toggleTodo(todo.id)}
 />
 <span>{todo.text}</span>
 <button onClick={() => deleteTodo(todo.id)}>Delete</
button>
 </li>
 ))}
 </ul>

 <div className="footer">
 <span>{activeCount} items left</span>
 {todos.some(t => t.completed) && (
 <button onClick={clearCompleted}>Clear completed</button>
 )}
 </div>
 </div>
 )
}

export default TodoApp
```

USER DASHBOARD

Dashboard with data fetching and context:

```
import { createContext, useContext, useState, useEffect } from
"react"
import { BrowserRouter, Routes, Route, Link, Navigate, useParams
} from "react-router-dom"

let AuthContext = createContext(null)

function AuthProvider({ children }) {
 let [user, setUser] = useState(null)
 let [loading, setLoading] = useState(true)

 useEffect(() => {
 let saved = localStorage.getItem("user")
 if (saved) {
 setUser(JSON.parse(saved))
 }
 setLoading(false)
 }, [])

 function login(userData) {
 setUser(userData)
 localStorage.setItem("user", JSON.stringify(userData))
 }

 function logout() {
 setUser(null)
 localStorage.removeItem("user")
 }

 if (loading) return <p>Loading...</p>

 return (
 <AuthContext.Provider value={{ user, login, logout }}>
 {children}
 </AuthContext.Provider>
 )
}
```

```
function useAuth() {
 return useContext(AuthContext)
}

function ProtectedRoute({ children }) {
 let { user } = useAuth()
 if (!user) return <Navigate to="/login" />
 return children
}

function Login() {
 let [email, setEmail] = useState("")
 let [password, setPassword] = useState("")
 let { login, user } = useAuth()

 if (user) return <Navigate to="/dashboard" />

 function handleSubmit(e) {
 e.preventDefault()
 login({ id: 1, email, name: email.split("@")[0] })
 }

 return (
 <form onSubmit={handleSubmit}>
 <h1>Login</h1>
 <input
 type="email"
 value={email}
 onChange={(e) => setEmail(e.target.value)}
 placeholder="Email"
 required
 />
 <input
 type="password"
 value={password}
 onChange={(e) => setPassword(e.target.value)}
 placeholder="Password"
 required
 />
 <button type="submit">Login</button>
```

```
 </form>
 )
}

function Dashboard() {
 let { user, logout } = useAuth()
 let [stats, setStats] = useState(null)
 let [loading, setLoading] = useState(true)

 useEffect(() => {
 setTimeout(() => {
 setStats({
 posts: 42,
 followers: 1234,
 following: 567
 })
 setLoading(false)
 }, 1000)
 }, [])

 return (
 <div>
 <header>
 <h1>Dashboard</h1>
 <span>Welcome, {user.name}</span>
 <button onClick={logout}>Logout</button>
 </header>

 <nav>
 <Link to="/dashboard">Overview</Link>
 <Link to="/dashboard/posts">Posts</Link>
 <Link to="/dashboard/settings">Settings</Link>
 </nav>

 {loading ? (
 <p>Loading stats...</p>
 ) : (
 <div className="stats">
 <div className="stat">
 <h3>{stats.posts}</h3>
```

```
 <p>Posts</p>
 </div>
 <div className="stat">
 <h3>{stats.followers}</h3>
 <p>Followers</p>
 </div>
 <div className="stat">
 <h3>{stats.following}</h3>
 <p>Following</p>
 </div>
 </div>
 )}
 </div>
 )
}

function App() {
 return (
 <AuthProvider>
 <BrowserRouter>
 <Routes>
 <Route path="/login" element={<Login />} />
 <Route
 path="/dashboard/*"
 element={
 <ProtectedRoute>
 <Dashboard />
 </ProtectedRoute>
 }
 />
 <Route path="/" element={<Navigate to="/dashboard" />} />
>
 </Routes>
 </BrowserRouter>
 </AuthProvider>
 )
}

export default App
```

SHOPPING CART

E-commerce cart with context and reducer:

```
import { createContext, useContext, useReducer, useMemo } from
"react"

let CartContext = createContext(null)

function cartReducer(state, action) {
 switch (action.type) {
 case "ADD_ITEM": {
 let existing = state.items.find(i => i.id === action.payload.id)

 if (existing) {
 return {
 ...state,
 items: state.items.map(item =>
 item.id === action.payload.id
 ? { ...item, quantity: item.quantity + 1 }
 : item
 )
 }
 }

 return {
 ...state,
 items: [...state.items, { ...action.payload, quantity: 1 }]
 }
 }

 case "REMOVE_ITEM":
 return {
 ...state,
 items: state.items.filter(i => i.id !== action.payload)
 }
```

```
 case "UPDATE_QUANTITY":
 return {
 ...state,
 items: state.items.map(item = >
 item.id = = = action.payload.id
 ? { ...item, quantity: action.payload.quantity }
 : item
 )
 }

 case "CLEAR_CART":
 return { ...state, items: [] }

 default:
 return state
 }
}

function CartProvider({ children }) {
 let [state, dispatch] = useReducer(cartReducer, { items: [] })

 let value = useMemo(() = > ({
 items: state.items,
 addItem: (item) = > dispatch({ type: "ADD_ITEM", payload: item
}),
 removeItem: (id) = > dispatch({ type: "REMOVE_ITEM", payload:
id }),
 updateQuantity: (id, quantity) = >
 dispatch({ type: "UPDATE_QUANTITY", payload: { id, quantity } }),
 clearCart: () = > dispatch({ type: "CLEAR_CART" }),
 total: state.items.reduce((sum, item) = > sum + item.price *
item.quantity, 0),
 itemCount: state.items.reduce((sum, item) = > sum +
item.quantity, 0)
 }), [state.items])

 return (
 <CartContext.Provider value={value}>
 {children}
 </CartContext.Provider>
```

```
 )
}

function useCart() {
 return useContext(CartContext)
}

let products = [
 { id: 1, name: "Product A", price: 29.99 },
 { id: 2, name: "Product B", price: 49.99 },
 { id: 3, name: "Product C", price: 19.99 },
 { id: 4, name: "Product D", price: 99.99 }
]

function ProductList() {
 let { addItem } = useCart()

 return (
 <div className="product-list">
 <h2>Products</h2>
 <div className="products">
 {products.map(product => (
 <div key={product.id} className="product-card">
 <h3>{product.name}</h3>
 <p>${product.price.toFixed(2)}</p>
 <button onClick={() => addItem(product)}>
 Add to Cart
 </button>
 </div>
 ))}
 </div>
 </div>
 )
}

function Cart() {
 let { items, removeItem, updateQuantity, clearCart, total } =
useCart()

 if (items.length === 0) {
```

```
 return <p>Your cart is empty</p>
 }

 return (
 <div className="cart">
 <h2>Shopping Cart</h2>
 <ul>
 {items.map(item => (
 <li key={item.id}>
 <span>{item.name}</span>
 <span>${item.price.toFixed(2)}</span>
 <input
 type="number"
 min="1"
 value={item.quantity}
 onChange={(e) => updateQuantity(item.id,
parseInt(e.target.value))}
 />
 <span>${(item.price * item.quantity).toFixed(2)}</span>
 <button onClick={() => removeItem(item.id)}>Remove</
button>
 </li>
 ))}
 </ul>
 <div className="cart-footer">
 <strong>Total: ${total.toFixed(2)}</strong>
 <button onClick={clearCart}>Clear Cart</button>
 <button className="checkout">Checkout</button>
 </div>
 </div>
 )
}

function CartIcon() {
 let { itemCount } = useCart()

 return (
 <div className="cart-icon">
 Cart ({itemCount})
 </div>
```

```
 )
}

function ShoppingApp() {
 return (
  <CartProvider>
  <header>
  <h1>Shop</h1>
  <CartIcon />
  </header>
  <main>
  <ProductList />
  <Cart />
  </main>
  </CartProvider>
 )
}

export default ShoppingApp
```

WHAT YOU BUILT

You built three complete React applications:

Todo App:
- State management with useState
- localStorage persistence
- Filtering and CRUD operations

User Dashboard:
- Authentication with context
- Protected routes
- Data fetching

Shopping Cart:
- Context with reducer
- Complex state updates

- Derived values

WHAT COMES NEXT

You now understand React fundamentals:

- What React is (Book 16, Chapter 1)
- Setting up React (Book 16, Chapter 2)
- JSX basics (Book 16, Chapter 3)
- Components (Book 16, Chapter 4)
- Props (Book 16, Chapter 5)
- State with useState (Book 16, Chapter 6)
- Handling events (Book 16, Chapter 7)
- Conditional rendering (Book 16, Chapter 8)
- Lists and keys (Book 16, Chapter 9)
- Forms and inputs (Book 16, Chapter 10)
- useEffect hook (Book 16, Chapter 11)
- Custom hooks (Book 16, Chapter 12)
- useRef hook (Book 16, Chapter 13)
- useContext hook (Book 16, Chapter 14)
- Component lifecycle (Book 16, Chapter 15)
- Styling components (Book 16, Chapter 16)
- React Router (Book 16, Chapter 17)
- Fetching data (Book 16, Chapter 18)
- Error boundaries (Book 16, Chapter 19)
- Performance optimization (Book 16, Chapter 20)

Book 17 teaches advanced React. You will:

- Use useReducer for complex state
- Learn advanced patterns
- Build reusable component libraries
- Master performance techniques
- Create production-ready apps

You finished Book 16. You build React apps. Now you master React.